Institute for Software Research

University of California, Irvine

A Survey of Software Engineering Educational Delivery Methods and Associated Learning Theories

Emily Oh Navarro
University of California, Irvine
emilyo@ics.uci.edu

April 2005

ISR Technical Report # UCI-ISR-05-5

Institute for Software Research
ICS2 210
University of California, Irvine
Irvine, CA 92697-3425
www.isr.uci.edu

www.isr.uci.edu/tech-reports.html

A Survey of Software Engineering Educational Delivery Methods and Associated Learning Theories

Emily Oh Navarro
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
emilyo@ics.uci.edu

ISR Technical Report # UCI-ISR-05-5

April 5, 2005

Abstract:

Software engineering education has acquired a notorious reputation for producing students that are ill-prepared for being productive in real-world software engineering settings. Although much attention has been devoted to improving the state of affairs in recent years, it still remains a difficult problem with no obvious solutions. In this paper, I attempt to discover some of the roots of the problem, and provide suggestions for addressing these difficulties. A survey of software engineering educational approaches is first presented. A categorization of these approaches in terms of the learning theories they leverage then reveals a number of deficiencies and potential areas for improvement. Specifically, there are a number of underutilized learning theories (Learning through Failure, Keller's ARCS, Discovery Learning, Aptitude-Treatment Interaction, Lateral Thinking, and Anchored Instruction), and the majority of existing approaches do not maximize their full educational potential. Furthermore, the approaches that engage the widest range of learning theories (practice-driven curricula, open-ended approaches, and simulation) are also the most infrequently used. Based on these observations, the following recommendations are proposed: Modify existing approaches to maximize their educational potential, design new approaches to address under-utilized learning theories, enhance the most promising approaches to make them more useful and effective, perform more formal and frequent evaluations of software engineering educational approaches, and frame software engineering education research in the context of learning theories .

A Survey of Software Engineering Educational Delivery Methods and Associated Learning Theories

Emily Oh Navarro
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
emilyo@ics.uci.edu

ISR Technical Report # UCI-ISR-05-5

April 5, 2005

1. Introduction

As software becomes more and more ubiquitous and foundational to an increasingly large part of modern society, it is not surprising that the attention given to software engineering education in recent years has also rapidly increased. This rising focus is evident in the emergence of entire conferences dedicated to the subject [7, 9], the Software Engineering body of Knowledge (SWEBOK) project [68], special journal issues centered around the topic [1, 4, 19], and special "software engineering education" tracks at major conferences [2, 3, 5, 6, 8]. Judging from the approaches published in these venues, it is clear that the overwhelmingly common basis for software engineering educational approaches consists of the following: a series of lectures in which concepts and theories are taught; and an associated small software engineering project that the students must complete in an attempt to put these concepts into practice. Although each educator's particular approach differs somewhat, common to nearly all of them are these two components.

Despite all of the attention, it is clear that much is still lacking in the way future software engineers are educated during their university education. Those who end up hiring new graduates frequently complain that they come into the workforce severely unprepared for practicing software engineering in a real-world setting, and the employers themselves must expend an enormous amount of time and effort to train them adequately [31, 38].

Educators have also acknowledged that there is a problem [41, 82, 89, 118]. Most seem to agree that it lies in the fact that the experience gained in the project component of software engineering curricula does not adequately reflect the experience in a real-world setting. In a 2001 IEEE Computer article, Bertrand Meyer pointed out that typical educational software engineering projects are "insufficient preparation for the true challenges of professional software development, which involve large systems (rather than the relatively small endeavors of academic projects), large groups (seldom the case in a university setting), modifying someone else's legacy system (rather than building a solution from scratch), and dealing with end users (rather than a professor and a grader)" [91].

Similar ideas were reflected in Mehdi Jazayeri's keynote talk at the 2004 Automated Software Engineering Conference [70]. He noted that class projects are by necessity typically constrained in the following ways: the instructor specifies the requirements, predetermines the infrastructure and development environment, fixes the schedule, trims the project to a manageable size, and ensures that there are no compatibility or legacy requirements. He acknowledges that, while these constraints are necessary for practical reasons, they are also the exact reasons why these projects grossly misrepresent real-world software engineering situations.

Although there is some consensus on what the problem is, there is little agreement on how to address the problem. Members of the software engineering educational community have proposed numerous approaches to making class projects more closely resemble the real world, as well as others that address the problem in various ways. But so far, no single approach (or set of approaches) has been accepted as a sufficient solution to the problem. The question is then, why is it so difficult to teach software engineering? Why are so many experienced, intelligent educators having such a hard time figuring out how to prepare future software engineers for the real world? What are the problems with current software engineering education delivery methods? Can we address these problems? If so, how?

In this paper I will explore these questions and attempt to provide some answers to them by surveying the various software engineering educational approaches and viewing them in light of well-known general educational theories. Specifically, I will describe the main categories of approaches, present the major relevant educational theories, investigate which theories these approaches have sufficiently exploited and which they have not, and, from this, make recommendations for ways in which we can more effectively educate future software engineers. It is important to note that this particular approach to addressing the problems of software engineering education will only address *part* of what makes software engineering education sub-optimally effective, namely, the ineffectiveness of delivery methods based on the learning theories they leverage (or fail to leverage). Undoubtedly, there are other factors that also contribute to the ineffectiveness of software engineering education, including many that may be intrinsic to the discipline of software engineering itself. This survey does not focus on such factors, but rather aims specifically to make recommendations for how to make software engineering education delivery methods more effective by leveraging more learning theories in their usage. This particular approach, evaluating delivery methods in terms of learning theories, was chosen because it is a common practice in other educational domains but has not yet been attempted in software engineering education.

The remainder of this paper is organized as follows: In Section 2, I describe and categorize the major approaches to software engineering education. In Section 3, I present a set of well-known learning theories that are especially applicable to software engineering education. Section 4 introduces a categorization of the software engineering educational approaches in terms of the learning theories they leverage. I provide key observations and recommendations for software engineering education in Section 5, discuss related work in Section 6, and conclude in Section 7.

2. Software Engineering Educational Approaches

In surveying the software engineering educational literature, it is clear that nearly every approach to teaching the subject is based on the same two components: lectures, in which software engineering theories and concepts are presented; and projects, in which students must work in groups to develop a (generally small) piece of software. However, judging from the dissatisfaction of industrial organizations that hire recent graduates (mentioned previously), it is clear that this approach is not sufficiently preparing future software engineers for jobs in the real world.

The academic community has recognized this problem and, in response to it, has created a wealth of innovations that build on the standard lectures plus project approach. These approaches fall into three major categories. The first involves attempts to make the students' project experience more closely resemble one they would encounter in the real world. The second category is simulation approaches, in which educators have students practice software engineering processes in a (usually) computer-based simulated environment. The final category includes approaches that teach one or more specific subjects a particular instructor feels are currently missing (e.g., usability testing or formal methods), and are crucial to effectively educating the students. It should be noted that some of these approaches span more than one category, and quite often a number of approaches from these different categories are combined in one course. It should also be noted that the purpose of this survey (to answer the question of why software engineering education in general is less-than-ideally effective) has certain implications for the scope of the approaches presented here. I incorporate only overall, high-level delivery methods that have been stated to be motivated expressly by the need to better prepare students for real-world software engineering careers.[1] I do not include approaches that deal with the details of how individual topics within software engineering are taught, such as how to teach the various protocols for white box testing, or UML notation. The focus instead is on approaches that aim to make the software engineering educational experience as a whole more effective, and in which the papers describing these approaches have expressly stated their motivation to be the need for better preparing future software engineers for the real world.

The remainder of this section describes these categories and their approaches in greater detail.

2.1 Adding Realism to Class Projects

It is clear from looking at the software engineering literature that the most common method of improving the educational experience involves modifying certain aspects of the class project to make it more closely resemble the experience students will face in their future software engineering careers. As the academic environment differs so greatly from the industrial, there are numerous angles from which educators have approached this issue, or aspects of the academic environment that they have tried to make more realistic.

[1] Throughout this paper, "the real world" should be taken to mean an industrial career or environment. These are the types of careers that the majority of graduates will find themselves in, and therefore all approaches surveyed are explicitly motivated by the need to better prepare students for the industrial environment.

2.1.1 Industrial Partnerships

Perhaps the most popular and practical way of making a class project more realistic is by partnering with an industrial participant. This type of approach varies in the level of involvement the industrial organization has with the project. (Note that some use a combination of the following approaches.) The lowest level of involvement consists of obtaining existing software developed by real-world organizations and requiring the students to modify that software [55]. A slightly more involved approach is putting the industrial participant in an advisory role, as they work with the instructor to design the project [20]. In other approaches, the industrial participant takes a more hands-on role with the students, holding seminars on special topics, providing guest lectures, offering feedback on their work from an industrial perspective, "pretending" to be a real customer, or working with the students as a mentor [113, 140]. Alternatively, in what could be termed as the "case study approach", the industrial organization provides the instructor with actual software engineering problems that they have solved in the past, or are solving in parallel, for the students to work on as exercises [56, 77, 79, 120, 140]. In such cases the industrial participant will also often "play" the customer role. In the most involved and most common type of approach, the industrial representative is an actual customer [52, 61, 62, 76, 79, 85, 129]. In such situations the students are provided with the experience of working on real-world applications that will potentially be used by real-world customers. Some of the benefits of industrial participation-type approaches that have been reported are: Higher motivation and satisfaction for the students, greater ease with which students obtain jobs after graduation, the opportunity for them to learn a real application domain, and a greater appreciation of quality and why it is important on the part of the students.[2]

2.1.2 Maintenance/Evolution-based Projects

A second type of approach to making class projects more realistic is the maintenance/evolution approach, in which students modify or enhance an existing software system rather than build one from scratch. This approach seems to come in two main varieties: In the first one, the maintenance project is ongoing over a number of semesters or quarters, and each class extends and builds on the previous classes' modifications [88, 114, 116, 137]. In the second type of approach, the piece of software being modified is unique to that particular class and/or semester [15, 55, 72, 101, 103]. These approaches generally argue that, since the majority of real-world projects are maintenance projects, students will be better prepared for the real world by becoming familiar with these types of projects during their university education.

2.1.3 Team Composition

While some educators focus on the nature of the project (e.g., maintenance-based or real-world), others focus on the nature and composition of the student teams that work on the project, to make them more closely mirror the team dynamics in real-world software engineering situations. One such approach is [113], which could be termed the "long-term teams" approach. In this approach students are grouped into teams during their freshman

---

[2] When describing the "claimed benefits" of each approach, I am referring to those that are claimed in the literature describing the approach. Most of these are not empirically proven phenomena, but rather claims based on anecdotal evidence.

year, and work in these same teams throughout their entire four-year program. Coupled with this is a series of projects that start out relatively simple in the earlier years, and become increasingly more complex as the students advance in the curriculum. This approach is designed to mimic the fact that teams in the real world generally work together for a period of time longer than just a quarter or semester, and to illustrate the dynamics of such long-term cohesion. An additional goal of this approach is to better train students in team skills—students in these long-term teams are forced to figure out ways to work together since they are "stuck with" each other for four years. A team that exists for only a semester or quarter knows that their team will soon be disbanded, so is perhaps less motivated to invest in making it work.

Another approach that focuses on making team dynamics more realistic is purposely making the teams very large. While academic project teams are normally composed of approximately three to six students, these approaches instead put the entire class on one development team [22, 23, 94]. While the students may take different roles in the process, they all work on the same project. This type of approach claims a number of benefits: First, it better exemplifies the size of teams in which students will work in their future careers. In the process, issues and dynamics that are unique to large groups are demonstrated. Second, it illustrates a more realistic control dynamic: No longer can any one person (student or instructor) be in control of, or even completely understand the entire project—they must truly depend on others to get the job done. Finally, it allows them to work on a project of greater (and hence, more realistic) size than would be feasible in a small team.

Another type of approach that focuses on the realism of team composition involves composing teams of students from different backgrounds and/or situations. The simplest of these is one in which the team members are all Computer Science students, but where each team consists of students from different classes within Computer Science (e.g., Human Computer Interaction, Databases, Software Engineering, Communication Technology) collaborating on the same project [121]. In this approach, students from each class work on the particular aspect of their project that pertains to the course they are taking. A slightly more diverse composition occurs when the teams are made up of students from different majors or degree programs within the same university [43, 90]. For instance, in one such project, English students did the technical writing, Marketing students did the market analysis, and Computer Science students wrote the code [90]. This "diversity" approach is used to try to prepare students for the real world in which they will work with people with different strengths and skills.

An alternate take on the "diversity" approach is one that combines students that are geographically distributed, coming from different universities [26, 27], and sometimes even from different countries [50]. In such cases the students never meet face-to-face, but instead must rely heavily upon groupware technologies in order to successfully collaborate. Such approaches argue that because software engineering in the real world is becoming increasingly distributed and global, exposing students to such situations early helps to better prepare them for their future careers.

The final category of approach that focuses on the realism of team dynamics involves dictating a well-defined, hierarchical team structure that the students must adhere to while completing their project [13, 52, 129]. In these approaches the students and teaching staff engage in a strict "corporate-style" interaction, simulating the group dynamics of

a real-world organization. Each student has a well-defined role (e.g., configuration manager, team lead, quality assurance engineer) and, in most cases, explicit protocols for communication and task management are defined. An example of an organizational chart from one of these types of approaches is shown in Figure 1. Each one of these roles is filled by one or more students, and each role has specific tasks for which they are responsible. It can be seen that this particular course had five different projects. One student is assigned to be the manager of the entire group, and their role is to balance the project-wide resources throughout the different teams, manage group-wide assignments such as presentations, and manage the support teams. For each project team, there is one project lead, one technical lead, and a number of engineers, depending on the project. Each of these project teams utilizes the services of the support team, which has specific people assigned to quality assurance, configuration management, process engineering, tools, and documentation. Each member of the support team plays a dual role on one of the project teams. Approaches like these are designed to better prepare students for the project team structures they will encounter in the real world. As a side result, these approaches also claim to foster better communication and produce higher quality software.

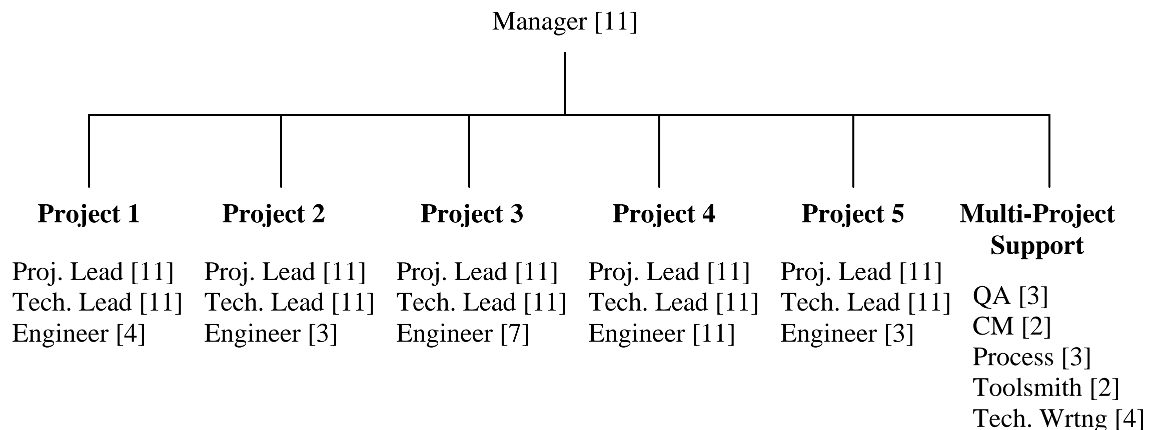

Figure 1: An Example of a Student Project Team Organizational Chart (adapted from [129]).

2.1.4 Non-Technical Skills

Along the same lines as the team dynamics-based realism approaches, some educators focus their software engineering class projects specifically on non-technical skills such as communication, group process, interpersonal competencies, project management, and problem solving [56, 57]. These types of approaches place more emphasis on social and logistical tasks typical in software projects, such as understanding the customer's domain and requirements, working in a team, organizing the division of work, and coping with time pressure in hard deadlines. As a result, the actual technical procedures of software engineering, such as writing code and designing software, are de-emphasized. For example, in [57] students are required to complete (and report upon) in-class team-building activities that emphasize communication, leadership, group problem solving, and decision making. These approaches identify such "soft" skills as what are most lacking in

university graduates, and hence argue that this kind of emphasis is crucial for their education.

2.1.5 Open-Endedness/Vagueness

In contrast to these highly structured approaches in which students are assigned specific tasks to complete in a specific way, others have tried to mimic the common less-structured real-world software engineering situations by making the project purposely open-ended and/or vague. There are two main ways in which this is done. In the first type of approach, the open-endedness comes in the requirements stage—requirements are made purposely unclear or indefinite, and the students must form most of them on their own [45, 94]. This approach is designed to give students the pseudo-experience of new product development based on market research. In the second type of approach, the *process* is unspecified, requiring the students to define their own process (sometimes with help from the instructor) [32, 58, 94, 116, 134] (note that [94] incorporates vagueness in *both* requirements and process). This is meant to give the students experience in not only following a process, but in designing the process that they follow. On top of these benefits, the open-ended approaches also claim that they are more motivating to students (which results in higher quality work on their part) since their focus is on *creating* rather than on following someone else's directions.

2.1.6 Practice-Driven Curricula

All of the approaches presented thus far have been designed to be used in conjunction with lectures, in which the concepts that the students are meant to be putting into practice in their projects are presented. In contrast, a somewhat radical approach that has been used is a "practice-driven" approach in which the curriculum is largely project-based [63, 97, 129]. In these approaches, lectures are used only as supporting activities. These approaches argue that theory is something that cannot be taught in a lecture, but instead must be built in each individual through experience and making mistakes. These approaches are designed to provide students with ample opportunity to make and learn from their mistakes, as they will undoubtedly do in the real world, albeit there with much more severe consequences. In one of these approaches, they conducted follow-ups with their students' employers after they graduated, which showed that these students needed less internal training than others in order to become productive in the workplace [97].

2.1.7 Sabotage

Another approach that encourages learning through mistakes or failure, although more explicitly, is deliberate sabotage. In this approach, the instructor purposely sets the students up for failure by introducing common real-world complications into projects, the rationale being that students will then be prepared when these situations occur in their future careers. Both [55] and [47] describe projects in which the instructor played a number of "dirty tricks" on the students during development. Some of the "dirtier tricks" included providing inadequate specifications, instructing the customer to be purposely uncertain when describing their needs, and purposely crashing the hardware just before a deadline. Table 1 presents all twenty of the tricks, including the underlying skills that the authors claim each trick teaches. The last column in the table indicates whether or not the particular trick is feasible in a typical academic setting.

2.1.8 Re-enacting Project Failures

A more severe way of purposely setting students' projects up for failure is presented in [21]. In this approach, the instructor assigned projects that had been known to fail in the past due to software process problems. In all cases, the students also failed, providing a perfect opportunity for the instructor to explain the rationale behind the best practices of software processes, as well as a way for the students to learn the consequences of not following these practices firsthand.

Table 1: The "Twenty Dirty Tricks" and the Skills they are Meant to Promote (adapted from [47]).

	Problem under-standing	People handling skills	Negotiation skills	Compromise skills	Planning skills	Adaptability	Quality under-standing	Organizational skills	Design skills	Possible at a university?
Inadequate specification	X	X	X					X		X
Make assumptions wrong	X		X				X			X
Uncertain customer	X	X	X							X
Change requirements	X		X		X	X			X	X
Conflicting requirements	X		X	X						X
Conflicting customer ideas	X	X	X	X		X				X
Different personalities	X	X	X			X				X
Ban overtime					X			X		
Additional tasks					X	X		X		?
Change deadlines			X	X	X	X				?
Quality inspections					X		X	X		X
Different truths		X				X				X
Change the teams		X			X	X	X	X		
Change working procedures		X				X		X		X
Upgrade the software					X	X				?
Change the hardware					X	X				X
Crash the hardware					X	X		X		
Slow the software					X	X		X		
Disrupt the file store					X	X	X	X		
Say "I told you so"	X	X	X	X	X	X	X	X	X	X

2.2 Simulation

While a majority of the software engineering educational approaches have focused on adding realism to class projects, a number of others have argued that the only feasible way to provide students with the experience of realistic software engineering processes within the academic environment is through simulation, used in conjunction with lectures and projects. While these approaches vary in terms of the processes they simulate and their specific purpose, they are all designed to allow students to practice and participate in software engineering processes on a larger scale and in a more rapid manner than can be feasibly done through an actual project. Judging from the relatively low number of publications in this area compared to other approaches (see Section 5), simulation is perhaps the most under-explored approach in software engineering education.

2.2.1 Industrial Simulation in the Classroom

The most straightforward type of approach to educational software engineering simulation is taking a software process simulator that is designed for industrial use and putting it in the classroom [37, 100]. These simulators are usually used for predicting the effects of

process changes and decisions on the eventual outcome of a project. The models run in these simulators are generally based strictly on empirical data. They typically have a non-graphical interface (meaning they display a set of gauges, graphs, and meters rather than characters and realistic surroundings) and a relatively low level of interactivity, taking a set of inputs such as man power, project size, and/or process plan, and outputting a set of results, such as budget, time, and defect rate. Figure 2 presents an example of one such simulator's user interface. In this particular interface, the user can see such data as percent complete and schedule pressure in the form of gauges, and other information, such as elapsed man hours and remaining hours, as numerical values. Through the buttons on the interface, they also have the ability to view graphs of this data, as well as start, stop, pause, and resume the simulation.

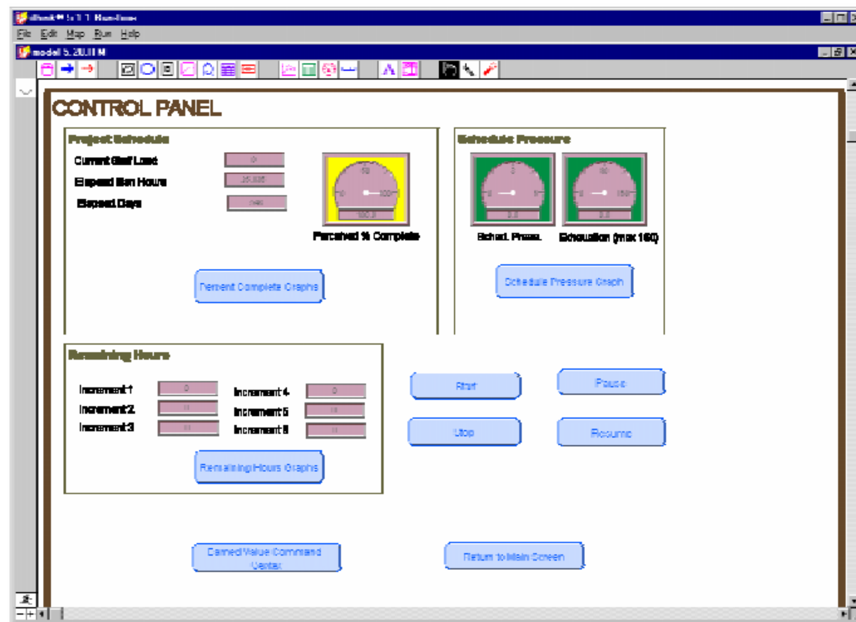

Figure 2: User Interface of an Industrial Simulator used in the Classroom [37].

When these simulators are used in the classroom, students are usually given an assignment involving the simulator, such as "simulate different life cycles and compare the results"; or "simulate this process with and without the use of code inspections, and note the difference in resulting defect rates." [37] has been used in the following incremental process: students are given simple simulations to begin with, which become increasingly more complex as their knowledge of the simulation and the model increases. These highly realistic simulations are designed to illustrate to students, using real-world data, the overall life cycle and project planning phenomena of software engineering.

2.2.2 Game-Based Simulations

Other educational software engineering simulations also simulate entire life cycles, but rather than use an industrial simulator, they do so in a game-based environment [18, 46, 49, 95, 117]. In these software engineering simulation games, the player is generally given a task to complete in the simulated world, such as "complete project X within 10 months, at or below a budget of $500,000, with a defect rate of less than 10%." The player will then interact with the game on a step-by-step basis to drive the simulation in order to complete the task. Some of these simulation games have graphical user interfaces in which the simulated physical surroundings are displayed, such as The Incredible Manager [46], SimSE [95], and OSS [117] (see Figure 3). Others, like SESAM [49], provide only textual output. Problems and Programmers [18] is a physical card game simulation that uses humorous descriptions, card illustrations, and unexpected situations to immerse the player in the process. With the exception of SESAM and SimSE, all of the other simulations come with one hard-coded simulation model, and do not support the development and simulation of other models. SESAM has also been used with an accompanying dialog and reflection session at the end of the simulation, in which students reflect on their experience and the lessons they have learned, and discuss this with a tutor or instructor [84]. The general hope in the game-based approach is that the additional enjoyment provided by the game features and dynamics will make learning about the particular software engineering process being modeled more memorable, and hence, more effective.

2.2.3 Group Process Simulations

While most educational software engineering simulations are used to simulate entire processes (or sub-processes), others are designed to specifically support structured group discussion and interaction processes [96, 125]. In these cases, the student engages in a discussion in which some or all of the other participants are simulated. The structured discussions being simulated are those that are typically present in software engineering

Figure 3: Graphical User Interface of OSS [117], a Software Engineering Simulation Game.

simulations, such as code inspections and requirements analysis meetings. For example, [125] describes an interactive video code inspection simulation that includes video and a natural language interface to immerse the user as a participant in the meeting. A screenshot of this simulation is shown in Figure 4. The user can choose which role they wish to play (moderator, reader, or recorder) in the inspection. The simulation incorporates an intelligent tutoring system that provides feedback to the student on their performance, both during and after the simulation. The requirements analysis meeting simulation described in [96] is embedded in a process that incorporates reflection, feedback, and discussion sessions as post-simulation activities. Such simulations are designed to give students experience in these kinds of discussions, which, these approaches argue, is one area in which new graduates are typically unprepared.

Figure 4: Simulated Characters in an Inspection Simulation [125].

2.3 Adding the "Missing Piece"

While the approaches discussed thus far have all mainly focused on changing the *manner* in which software engineering concepts are taught, there is another large school of thought that concentrates instead on changing the *content* of what is taught—in particular, these approaches believe that software engineering education is lacking in effectiveness due to the omission of one (or a few) important subject(s). What this "missing piece" is varies from approach to approach, but all generally believe that the addition of this subject to the curriculum (either to an existing course or as an entirely new and separate

course) will make the students' education much more complete, better preparing them for the real world.[3]

Some of these approaches believe that formality is underemphasized. Both [10] and [135] argue that formal methods should be taught more, while [41] proposes teaching traditional engineering education as part of a software engineering course.

Others believe that students should be taught a specific software process and be required to follow that process in their academic software engineering projects. These approaches teach such processes as the Personal Software Process (PSP) [33, 35, 65, 81, 83, 107, 110, 128-130, 138] or a variation of PSP [67, 131, 132], a Team Software Process (TSP) variety [127], [107], a Rational Unified Process (RUP) variety [52, 60], [108], or Extreme Programming (XP) [63, 119].

Still others propose that students should not only be required to follow a specific software process, but to also practice process engineering and project management techniques to create their own software processes, and use process improvement techniques to improve upon them [29, 58, 69]. For example, [69] presents an approach in which students must use a software process simulator to build, simulate, and improve upon the process used in a group project. Others take a broader approach, in which they require students to learn and utilize general project management techniques [86, 93, 139].

Rather than focus on process as a whole, other approaches focus on specific parts of the process, and specific techniques for performing that task. For example, [40] proposes teaching scenario-based requirements engineering, [131] includes extensive teaching of code reviews, and usability testing is emphasized in [136].

Some approaches, rather than teach software engineering in general, focus on a specific type of software engineering. Examples of this are [15], [122], and [103], which all emphasize maintenance- and evolution-based software engineering; [98] and [51], which focus on component-based software engineering; and [75], [78], and [44], which emphasize software engineering for real-time applications.

Still others believe it is certain non-technical aspects of software engineering that should be added to the software engineering curriculum. [57], [56], and [59] present class project setups that emphasize skills such as communication, group process, interpersonal skills, project management, and problem solving over the more typical technical abilities. [102] argues that software engineering students should receive more training in how to interact with various stakeholders. The importance of including a course in Human-Computer Interaction in a software engineering curriculum is argued for in [133] and [64]. Finally, [114] proposes that it is important to include the business aspects of software engineering, such as intellectual property, product marketing and distribution, and financial models.

2.4 Summary of Software Engineering Educational Approaches

It is clear that software engineering education is an important research area to which much attention has been given in recent years. Numerous educators are actively contributing their ideas and approaches to making the software engineering class project envi-

[3] All approaches thus far have dealt strictly with delivery methods. The "missing piece" approach is the only one that crosses the line into topics. However, it is included in this survey because of each of these approach's explicit statement of their motivation being the need to better prepare software engineers for the real world.

ronment more realistic, creating simulations in which students can freely experience simulated software engineering processes, and adding overlooked software engineering subjects to the curriculum. However, the presence of all these disparate approaches, as well as the lack of agreement on which one(s) are best and the continued dissatisfaction of industry, signifies that the problem of effectively educating future software engineers is by no means a solved one.

3. Learning Theories

When discussing and evaluating educational approaches, it is only appropriate that the discussion is tied back to the roots of educational theory. Learning theories are theories that describe how people learn. One of the main purposes of learning theories is their use as a guide in evaluating and modifying existing educational approaches, as well as in creating new ones. In this section, I will present and describe some of the most widely accepted and well-known learning theories that are relevant to the domain of software engineering education. This section will provide the background necessary for a categorization of software engineering educational approaches in terms of these learning theories, which will be presented in Section 4.

This particular set of learning theories was chosen according to two criteria: orthogonality and relevancy. In order to create an ideal framework for the categorization presented in Section 4, and to avoid redundancy, I aimed to include the more orthogonal theories that exist. There exists a great deal of overlap among learning theories, and there are several learning theories that encompass a number of others. In these cases, I either group theories that have the same basic idea (and explicitly mention and describe all of the ones that I group), and omit those that simply combine a number of theories. By necessity, however, there are still some theories that share common elements, but have some other distinguishing characteristic(s). I also omitted four other categories of learning theories. I did not include those that are very general cognitive theories (with no obvious application to software engineering education), such as cognitive dissonance, which deals with how one resolves conflicting notions in the mind. I also omitted those that are specific to other subjects (such as math, reading, or foreign language), those that are specific to irrelevant people groups (such as older adults or elementary school-aged children), and those that are simply not relevant to software engineering education (such as behaviorist theories).

3.1 Learning by Doing / Experiential Learning / Active Learning / Constructivism

It is well known that people learn a task best not only by hearing about it, but also by actually *doing* it. This is the basic premise of the Learning By Doing educational theory [16, 104, 109, 111, 112] (also known as / virtually identical to Experiential Learning). Often when a student hears about a skill or piece of knowledge in theory, the information appears straightforward, and they assume they have grasped it. However, when they actually attempt that task, they realize how difficult and complex the skill or piece of knowledge actually is. It is from this point that they truly begin to learn the material, as they discover by experience the complexities and depth of the subject. Alternately, a piece of knowledge can seem complex and difficult to grasp at first, until the learner puts that knowledge into practice, at which point it becomes clear and they begin to master it.

In either case, it is only by actively *doing* something with the knowledge that the student truly learns it.

A relevant subset of the Learning by Doing theory is the idea of Active Learning [12, 24]. This theory builds on Learning by Doing by specifying the "doing" tasks, namely, analysis, synthesis, and evaluation. Such tasks encourage students to not only do the task, but to also think about what they are doing, in order to gain a deeper level of knowledge and understanding.

Another related subset of Learning by Doing is Constructivism [17, 30, 87]. The basis of this theory is that knowledge is something that is built or constructed through experience, not something that can simply be told or conveyed. Constructivism is highly learner-centric, focusing less on the actions and thinking of the teacher and more on those of the learner—how they construct new concepts or ideas based on their past knowledge and current experience.

To sum up and tie these concepts together, the implication of these theories for educational approaches is the following: the learner should be provided with ample opportunity to actually *do* what they are learning about, not simply absorb the knowledge through a lecture, book, or some other medium; and in so doing, should be encouraged to reflect upon their actions through analysis, synthesis, and evaluation activities. An ideal example of this would be a situation in which one is learning the skill of writing fiction: An approach that takes advantage of these theories would not only have the student hear lectures about fiction-writing and read about how to write fiction; they would also actually write stories themselves, as well as analyze and evaluate their work and the work of others.

3.2 Situated Learning / Functional Context / Andragogy / Schema Theory

Situated Learning [12, 28, 54, 106, 124, 126] (also known as / virtually identical to Functional Context) is an educational theory that builds upon the Learning by Doing approach. However, while Learning by Doing focuses on the specific learning activities that the student performs, the Situated Learning educational theory is concerned with the environment in which the learning by doing takes place. In particular, Situated Learning is based on the belief that knowledge is situated, being in large part a product of the activity, context, and culture in which it is developed and used. Therefore the environment in which the student practices their newly learned knowledge should resemble, as closely as possible, the environment in which the knowledge will be used in real life. Situated Learning argues that typical school learning does not apply well to workplace situations because it does not provide a meaningful context in which skills are taught, and gives too much attention to theoretical explanation rather than to building true performance skills.

One of the main educational practices proposed by the Situated Learning theory is that of *cognitive apprenticeships* [36]. The idea of a cognitive apprenticeship is similar to that of a typical craft apprenticeship. However, its focus is to bring students into the *culture* of practice rather than simply engage them in the relevant activities. It does so by providing for them an authentic environment and authentic social interaction in which to practice their newly learned skills and knowledge.

It is worth mentioning two other educational theories that are implied in Situated Learning: Andragogy and Schema Theory. Andragogy [73] is a theory that is specific to

adult education and is based on two basic premises: 1) In order to learn effectively, adults need to know why they are learning something (e.g., provide examples of what happens if you do not follow the practices being taught); 2) In order to learn effectively, adults need to know that what they are learning has a clear effect on their ability to handle real-world situations. Andragogy has an obvious connection to Situated Learning in that allowing students to practice skills in environments that resemble the real world will show them how that piece of knowledge is useful in the real world.

Schema Theory [14] is also related, but applies to all ages and stages of education. Schema Theory views organized knowledge as a complex network of *schemata*, or abstract mental structures, that represent an individual's understanding of the world around them. Therefore the role of the teacher is to help learners build schemata and make connections between ideas by demonstrating how a piece of knowledge applies in the real world.

It is clear that the implication of these theories for educational approaches is to make the environment in which a skill or piece of knowledge is practiced resemble the environment in which the knowledge will be used in real life, thereby demonstrating the relevance and direct usefulness of that knowledge to real-world situations. For instance, a journalism student may hear and read a great deal about journalistic style, culture, and techniques, and may even put some of these techniques into practice by writing articles for their instructor to grade. However, the only "situated" way for the student to learn about journalism would be by writing for the school paper or doing an internship for a commercial magazine or newspaper.

3.3 Keller's ARCS Motivation Theory

Like Situated Learning, Keller's ARCS Motivation Theory [71] also focuses on motivating students to learn. However, rather than focusing on the physical environment in which they learn, Keller's ARCS Motivation Theory concerns itself with promoting certain feelings in the learner that motivate them to learn. In particular, these feelings are attention, relevance, confidence, and satisfaction.

- **Attention:** The attention and interest of the learner must be engaged. Proposed methods for doing so are: introducing unique and unexpected events; varying aspects of instruction; and arousing information-seeking behavior by having the learner solve or generate questions or problems.
- **Relevance:** Learners must feel that the knowledge is relevant to their lives. The theory suggests that knowledge be presented and practiced using examples and concepts that are relevant to learners' past, present, and future experiences.
- **Confidence:** Learners need to feel personal confidence in the learning material. This should be done by presenting a non-trivial challenge and enabling them to succeed at it, communicating positive expectations, and providing constructive feedback.
- **Satisfaction:** A feeling of satisfaction must be promoted in the learning experience. This can be done by providing students with opportunities to practice their newly learned knowledge or skills in a real or simulated setting, and providing positive reinforcements for success.

To summarize, Keller's ARCS Motivation theory suggests that attention, relevance, confidence, and satisfaction (ARCS) are the conditions that, when integrated, motivate an individual to learn.

3.4 Anchored Instruction / Problem-based Learning

Anchored Instruction [25] is another theory that deals with teaching techniques. In particular, Anchored Instruction says educators should center all learning activities around an "anchor"—a realistic situation, case study, or problem. Presentation of general concepts and theories should be kept to a minimum. Instead, Anchored Instruction believes that knowledge is best learned by exploration of these realistic case studies or problems.

Problem-based Learning [123] is a well-known subset of Anchored Instruction. It is more specific, however, in that it requires that the anchor be a real-world problem that students must work together in groups to solve. It places significant emphasis on the requirement that the problems be real-world—since they are usually ill-defined and unstructured (rather than "cut and dry" like many toy problems), they strongly promote critical thinking and problem-solving skills that will be invaluable to students' future careers.

An educational approach that leverages both Anchored Instruction and Problem-based Learning is frequently used in training medical students: Specifically, the course consists of nothing but presenting a group of students with a series of bizarre real-world cases (e.g., "A 93-year old woman appears to be pregnant"). For each problem, the group is asked to brainstorm the possible causes of the condition, do research, formulate hypotheses, and recommend a solution.

3.5 Discovery Learning

The Discovery Learning theory [12, 104] takes a similar approach to Anchored Instruction in that it believes that an exploratory type of learning is best. Discovery Learning is based on the idea that an individual learns a piece of knowledge most effectively if they discover it on their own, rather than having it explicitly told to them. This theory encourages educational approaches that are rich in exploring, experimenting, doing research, asking questions, and seeking answers.

An example of Discovery Learning would be a Chemistry curriculum that is driven by lab experiments—specifically, one that in which concepts are discovered through lab experiments, and lectures are used only to further explore these discovered concepts.

3.6 Learning through Failure

Along the same lines as the Discovery Learning theory is the Learning Through Failure theory [111]. This approach is based on the assumption that the most memorable lessons are those that are learned as a result of failure. Learning through failure also provides more motivation for students to learn, so as to avoid the adverse consequences that they experience firsthand when they do not perform as taught. Failure can also engage students, as they are motivated to try again in order to succeed. Proponents of the theory argue that students should be allowed to (and even set up to) fail to encourage maximal learning. For example, pilots-in-training begin their training in flight simulators (rather than real airplanes). The trainees are expected to crash land several times (but safely, as the crash is not real, but simulated) before they master the skill of landing an aircraft.

3.7 Learning through Dialogue / Conversation Theory

While most of the learning theories discussed so far focus mainly on the learner as independent and responsible for fostering their knowledge on their own (using the proper learning materials/activities), the Learning through Dialogue theory [39] gives the teacher a much more active and pivotal role in the learner's education. Learning through Dialogue suggests that dialogue between student and teacher is necessary for effective learning and retention. According to the theory, this dialogue should consist of the teacher encouraging reflection, assessing the student's aptitudes and learning style, and tailoring their teaching strategy accordingly. Learning through Dialogue is based on Conversation Theory [99], a learning theory that emphasizes the importance of the learner conversing with others about the subject matter in order to learn it effectively. Conversation Theory, however, is slightly more general, in that allows for the other party in the conversation to be anyone, not just the instructor.

3.8 Aptitude-Treatment Interaction

Like Learning through Dialogue, the Aptitude-Treatment Interaction [42] theory also recommends that the instructor take an active role in assessing the characteristics of the learner and modify their teaching style accordingly. Aptitude-Treatment Interaction focuses primarily on the aptitude of the learner, and states that the learning environment should be tailored to this particular characteristic: Specifically, low-ability learners need highly structured learning environments that incorporate a high level of control by the instructor, concrete and well-defined assignments, and specific sequences to follow for completing them. High-ability learners, on the other hand, tend to be more independent, which implies a less structured approach is more effective for this type of student.

3.9 Multiple Intelligences

Like the Aptitude-Treatment Interaction theory, the theory of Multiple Intelligences also deals with the diverse learning needs and styles of individuals. However, while Aptitude-Treatment Interaction is primarily concerned with the aptitude of the learner, the Multiple Intelligences [53] theory is instead focused on the particular learning modalities that are unique to each individual. In particular, the theory identifies seven different learning modalities: linguistic, musical, logical-mathematical, spatial, body-kinesthetic, intrapersonal (metacognition and insight), and interpersonal (social skills). Whenever possible, instruction should be individually tailored to each student to target the particular learning modalities that are most effective for them. For example, a highly musically-inclined student who is learning computer programming should first be introduced to the subject by programming a simple musical piece. In contrast, a student with strong spatial skills should have as their first assignment some type of graphics program.

3.10 Learning through Reflection

The theory of Learning through Reflection is primarily based on Donald Shon's work suggesting the importance of reflection activities in the learning process [115]. In particular, Learning through Reflection emphasizes the need for students to reflect on their learning experience in order to make the learning material more explicit, concrete, and memorable. Reflection activities encourage individuals to take a "meta" view of their

learning experience by encouraging them to explore their learning process, evaluate their performance, assess how their learning has prepared them for future experiences, and determine which skill or knowledge areas they lack. Some common reflection activities include discussions, journaling, or dialogue with an instructor [74]. Educational approaches that leverage the Learning through Reflection theory are rich in these types of activities.

3.11 Elaboration / Information Processing Theory

While Learning Through Reflection is primarily concerned with what individuals do with knowledge once they have received it, the theory of Elaboration [105] is focused on how that information is presented to the learner in the first place. In particular, it states that, for optimal learning, instruction should be organized in order of complexity, from least complex to most complex. Simplest versions of tasks should be taught first, followed by more complicated versions.

It is worth mentioning that the theory of Elaboration is based on another learning theory: that of Information Processing [92]. The Information Processing theory states that short-term memory, or attention span, can process no more than seven pieces of information at a time. In addition, it states that processing information in sequential steps is an essential cognitive sub-process of learning. Hence, these theories together imply that educational approaches should present information to students in small chunks that start out simple and become increasingly more complex.

3.12 Lateral Thinking

While the Elaboration theory deals with the size and type of information to be presented, the Lateral Thinking theory [48] is instead concerned with how students are encouraged to think about the information presented. Specifically, Lateral Thinking states that knowledge is best learned when students are presented with problems that require them to take on different perspectives than they are used to and practice "out of the box" thinking. The theory suggests that students be challenged to search for new and unique ways of looking at things, and in particular, these views should involve low-probability ideas that are unlikely to occur in the normal course of events. It is only through this type of relaxed, exploratory thinking that one can obtain a firm grasp on a problem or piece of knowledge.

The following anecdote (taken from [48]) demonstrates lateral thinking in action: A certain man owed money to a lender. This man convinced the lender to settle the debt based upon a random drawing: Two stones would be placed in a bag, one white and one black. The man's daughter would draw a stone. If she drew the white stone, the debt would be forgiven. If she drew the black stone, the lender would get to take the man's daughter for his wife. The man attempted to outsmart the lender and put two black stones in the bag. But his daughter was even smarter: When she drew a stone, she immediately dropped it on the ground, amongst the other stones naturally occurring there. Hence, it was immediately indistinguishable from these other stones. She then argued that the stone she drew must have been the opposite color from the one remaining in the bag, and she was able to prevent herself from being taken by the lender. This man's daughter solved an intractable problem through lateral thinking.

3.13 Summary of Learning Theories

In this section I have presented a basic background of learning theories that are applicable to software engineering education. Although these theories have distinguishing characteristics from each other, it is clear that they all share a common goal: To make learning more motivating, relevant, and effective, whether through the setup of the learning environment (Situated Learning, Anchored Instruction, Discovery Learning, Learning through Failure, Elaboration, Learning by Doing), the instructor's actions (Learning through Dialogue, Aptitude-Treatment Interaction, Multiple Intelligences), the learner's thought process (Learning through Reflection, Lateral Thinking), or any combination of these that achieve a desired result (Keller's ARCS).

4. Software Engineering Educational Approaches and Learning Theories

Obviously, there are both a wealth of innovative software engineering educational approaches, and a wealth of learning theories that can be leveraged in designing an educational approach. However, is the former taking maximal advantage of the latter? And could the answer to this question explain why, in spite of all of these innovations, software engineering education is still so difficult and ineffective in preparing students for the real world? I will now take a look at the approaches presented in Section 2 in light of the educational theories presented in Section 3 in order to address this question.

However, before doing this, it is necessary to state a basic assumption on which I will be making this analysis and the subsequent observations and recommendations. The assumption is the following: The more learning theories a method incorporates, the more effective it will be. This is not an empirically proven fact, but there exists an enormous body of literature in other educational domains that also bases their educational evaluations on this assumption. However, I do not claim that this assumption is necessarily true in all cases. This assumption is made with the following two caveats: First, it is more effective to choose a method that incorporates a number of learning theories naturally and coherently, rather than combining different methods that each utilize a different learning theory. This means that, if the desired effect is to incorporate x learning theories, it is more effective to choose Method A, which incorporates all x naturally rather than trying to combine Methods B, C, D, and E, which each incorporate a subset of the x learning theories. Such an approach would most likely be unfocused and confusing. The second caveat is that, for any given delivery method, there may be an optimal number of theories with a drop-off in effectiveness at some point. For instance, Method A might be optimal with x theories, but once it tries to incorporate (x + 1) theories, the effectiveness may start to decrease.

In Table 2, the analysis that is based on this assumption is presented. Table 2 is a matrix of software engineering educational approaches, and the learning theories that they leverage. There are a number of approaches within the Projects plus Realism category that take advantage of the same set of learning theories. These are all in the lower half of the table, and, since they leverage the same exact set of learning theories, they will not be discussed separately, but will instead be grouped together in the discussion throughout the remainder of this paper as the "Projects plus Realism (all others)" category. Some other approaches are also discussed together, if they are highly similar and/or employ the

Table 2: Software Engineering Educational Approaches and the Learning Theories they Leverage.

	Learning by Doing et. al.	Situated Learning et. al.	Keller's ARCS	Anchored Instruction	Discovery Learning	Learning Through Failure	Learning Through Dialogue	Aptitude-Treatment Interaction	Multiple Intelligences	Learning Through Reflection	Elaboration	Lateral Thinking
Industrial Partnership – Real Project	X	X	X				X/P			X/P		
Team Composition – Long-Term Teams	X	X					P			P	X	P
Open-Endedness – Requirements	X	X	X		X	X	P			P		
Open-Endedness – Process	X	X	X		X	X	P			P		
Practice-Driven	X			X	X	X	X/P	P	?	X/P	P	
Sabotage	X	X				X	P			P	P	
Re-enacting Failures	X	X				X	P			P	P	
Simulation – Industrial	X	X	X	P	X	X	P	P	?	P	X/P	P
Simulation – Game-Based	X	X	X	P	X	X	X/P	P	?	X/P	P	P
Simulation – Group Process	X	X	X	P	X	X	X/P	P	?	X/P	P	X/P
Missing Piece	X											
Industrial Partnership – Modify Real Software	X	X					P			P	P	
Industrial Partnership – Industrial Advisor	X	X					P			P	P	
Industrial Partnership – Involved Mentor	X	X					P			P	P	
Industrial Partnership – Case Study	X	X					P			P	P	
Maintenance/Evolution – Multi-semester	X	X					P			P	P	
Maintenance/Evolution – Single-semester	X	X					P			P	P	
Team Composition – Large Teams	X	X					P			P	P	
Team Composition – Different C.S. Classes	X	X					P			P	P	
Team Composition – Different Majors	X	X					P			P	P	
Team Composition – Different Universities	X	X					P			P	P	
Team Composition – Different Countries	X	X					P			P	P	
Team Composition – Team Structure	X	X					P			P	P	
Non-Technical Skills	X	X					P			P	P	

same learning theories (e.g., sabotage and re-enacting failures). An 'X' in the table indicates that there have been approaches within that category that have leveraged that theory, a 'P' represents that there is an obvious potential for that particular type of approach

to employ that learning theory (in and of itself, not combined with any other approach), but there have been no known cases of it. The presence of both an ‘X’ and a ‘P’ indicates that perhaps one or two approaches in the category have taken advantage of the theory, but most have not, so there is significant potential for further exploitation. A question mark denotes an open research question that is difficult to determine without further investigation.

4.1 Industrial Partnership – Real-world Project

A specific approach from the most popular of the adding-realism-to-class-projects category, the “Industrial Partnership” approach, fully leverages three of the ten learning theories. In particular, I refer to the specific approach in which students work on actual real-world projects. Only three of these five learning theories are utilized by all approaches in this category, while the other two are only used in one particular approach. First, these approaches allow students to “learn by doing”, in that they learn how to work on real projects by actually working on them. Second, they provide a learning environment that is more similar to that of the real world, namely, by the addition of the “real project” environmental factor (leveraging the Situated Learning theory). If we recall one of this approach’s claimed benefits from Section 2.1.1 (students learn a greater appreciation of quality and why it is important), it is clear that this benefit could be directly attributed to the situated nature of the approach: students exposed to a more realistic environment through a real-world project realize the importance of quality because quality is important in a real world with real customers. In an academic environment in which customers are usually “fake” (or nonexistent), quality is intrinsically not as crucial.

Third, these approaches promote Attention, Relevance, Confidence, and Satisfaction (Keller’s ARCS). As these approaches have also reported, a “real” project is taken more seriously by students, considered more relevant to their lives, and promotes a greater feeling of confidence, motivation, and satisfaction.

Learning through Dialogue and Learning through Reflection are two theories that are not as heavily utilized by the industrial partnership – real project approach. Specifically, it is only mentioned in the approach described in [129], which incorporates weekly one-on-one mentoring sessions with a “coach” that discusses each student’s performance with them, answers their questions, makes suggestions, and helps them reflect on their experience. This is a relatively simple component that could be added to the other approaches to take advantage of these two learning theories and thereby enhance their educational effectiveness.

4.2 Team Composition – Long-Term Teams

As described in Section 2.1.3, the “long-term teams” approach has students work in the same teams for their entire four-year program, on projects that start out simple in their first year and gradually become more complex as the years continue. This approach takes full advantage of three learning theories. First, as this approach claims, the students learn how to work in an unchanging team for a long period of time (a characteristic of real-world software engineering teams) by actually doing so (Learning by Doing). Furthermore, the fact that they do so for a long period of time suggests that this approach allows them more “doing” (with respect to practicing team skills) than some other approaches. Second, intrinsic to the educational effectiveness of this experience is the meaningful,

realistic context (long-term teams) in which these skills were learned (Situated Learning). Finally, because the projects start out simple and incrementally add complexity (elaborate), this approach also leverages the Elaboration theory.[4]

The theories of Learning through Reflection, Learning through Dialogue, and Lateral Thinking also have significant potential to be used with modified versions of this approach. First, both Learning through Reflection and Learning through Dialogue could be leveraged if the approach incorporated structured reflection and dialogue techniques, in which students could reflect upon and discuss their long-term team experience. Second, because the teams in this approach are together for such a long period of time, it would be an ideal situation for students to rotate roles and responsibilities in the projects, directly utilizing Lateral Thinking. Unlike a shorter-term team approach, these students could have the opportunity to get a good feel for a number of different roles, as they could potentially spend several months in each.

4.3 Open-endedness

As discussed in Section 2.1.5, an open-ended approach entails the instructor purposely leaving some parts of the project vague, either in terms of the requirements of the software or the process the students are to follow. This type of approach fully leverages five learning theories. First, Learning by Doing is an integral part of this approach: Students learn how to create and then follow a self-defined process by doing so, and also learn how to do market research to define their own requirements by doing so. Second, Situated Learning is employed, as the learning environment is made more situated, or realistic: In the process vagueness approach, the students must define their own process, which is a normal practice in real-world software engineering situations, in contrast to having an instructor specify all intermediate deadlines, deliverables, and procedures. In the requirements vagueness approach, they are put in the realistic situation of needing to do market analysis to define the requirements for a new product.

The third learning theory the open-ended approach leverages is Keller's ARCS. Looking back to Section 2.1.5, one of the claims of this type of approach is that it is more motivating to students and results in higher quality work, due to the fact that the students have a greater ownership of the project—they design their own requirements, process, or both, not simply follow the dictates of others. This claim fits squarely in the four tenets of Keller's ARCS theory:

- **Attention:** Clearly, the high level of motivation reported attests to the fact that the attention of the students was sparked and engaged with this approach.
- **Relevance:** An integral part of this approach is its portrayal of how things are done in the real world, in which such issues as requirements and process are often vague.
- **Confidence:** The fact that the students in these approaches were successful in producing high quality work is both evidence of, and a contributing factor to the confidence these students cultivated through the experience. Their increased

[4] It could be argued that the long-term team approach's utilization of the Elaboration theory should not be compared to others for the following reason: It is specific to a four-year time period, during which it is only natural that instruction become more advanced as students progress, while other approaches mainly refer to projects in general without specifying a time period. However, since this approach obviously and specifically leverages the Elaboration theory, it remains included in this comparison.

amount of ownership in the project and their subsequent success perhaps promotes even more confidence than a non-open-ended approach.

- **Satisfaction:** Again, the students' success in a project that they felt a great deal of ownership in is a factor that fosters a high level of satisfaction.

The open-ended approach also richly utilizes the Discovery Learning theory. As this theory states, an individual learns most effectively if they discover a piece of knowledge on their own, rather than having it explicitly told to them. An open-ended approach promotes this type of learning, as students are simply thrown into a situation in which they must make the vague concrete and the undefined defined, in the process discovering how to do market research to formulate requirements for a new product, or how to design and follow a process.

Learning through Failure is another theory that is directly leveraged by the open-ended approach. As students are thrown into a vague situation and required to simultaneously discover, develop, and employ their skills, they are bound to fail at times. But it is through these failures that they have some of the greatest opportunities to learn "what not to do" and "what they should have done."

Finally, this type of approach has potential to make use of Learning through Reflection and Learning through Dialogue if reflection and dialogue activities are incorporated. These activities could be particularly useful in analyzing, discussing, and reflecting on the students' mistakes and failures and gleaning all that can be learned from them.

4.4 Practice-Driven Curricula

A practice-driven curriculum, as described in Section 2.1.6, is one that is heavily based on projects and labs, and believes that true knowledge can only be built through experience, making mistakes, and learning from them (which this approach gives students plenty of opportunity to do). This approach encompasses four educational theories. It has a heavy Learning by Doing slant, in that virtually the entire approach consists of "doing" activities rather than passive activities such as listening to lectures and reading. It also directly implements the Anchored Instruction theory, which has a similar philosophy to that mentioned by this type of approach: As stated previously, Anchored Instruction believes that all learning activities should be centered around an "anchor" (in this case a project or a lab) that is actively explored and practiced. This, the theory states, is the way that knowledge is most effectively learned, not through presentations of general concepts and theories, which should be kept to a minimum. The related theory of Discovery Learning is also utilized by this approach—a practice-driven curriculum allows an individual to discover knowledge on their own through exploration and experimentation, rather than having it explicitly told to them. Finally, an integral part of this approach is Learning through Failure: Students are expected to fail several times as they attempt the projects and labs with little background knowledge or preparation, but this failure is the expected avenue to gaining true insight and knowledge.

There is also the potential for four other learning theories to be incorporated into a practice-driven approach. First, dialogue and reflection sessions could be added as part of the curriculum, in which the knowledge discovered in the anchor activities could be reflected upon and discussed, which would make it more explicit and perhaps more memorable. At the same time, these types of sessions might reveal knowledge that one had subconsciously discovered but was not consciously aware of yet (e.g., "I knew I had resolved

this conflict with my team, but I didn't realize that I discovered teamwork skills in the process"). Such activities have only been incorporated into one of these approaches ([129]).

Second, there may be the potential for the Aptitude-Treatment Interaction theory to be incorporated into this approach. In particular, its project-intensive nature makes it an ideal situation for varying the amount of structure for each student according to their aptitude: An instructor could give more open-ended projects and assignments to higher aptitude students, and more concrete and structured ones to those of lower aptitude. (This would not be possible with a lecture-based curriculum, as it is not feasible to tailor a lecture to each individual student's needs.) While there may be several obvious drawbacks to such an approach (namely, unfairness, and feelings of ineptitude on the part of those who are assigned more structured projects), the benefits claimed by the Aptitude-Treatment Interaction theory may warrant further investigation into this approach.

Along the same lines, the theory of Multiple Intelligences could also potentially be incorporated into this approach. Specifically, rather than changing the level of structure of the projects or assignments, their *content* could vary based on the learning modalities of each student. There is an obvious question of feasibility here however, which is why this particular box in the chart has a question mark rather than a "potential": Is it feasible for an instructor to assess the learning modalities of each student and design for them an appropriate project that takes advantage of these? Is there a way this process could be automated, perhaps with a quiz or tool that assesses learning modalities, and a standard set of projects for each modality?

Finally, this approach could also utilize the theory of Elaboration if the anchors (projects and labs) are assigned in order of increasing complexity. Although it could probably be safely assumed that the projects in these approaches are being organized in some sort of increasingly complex order, they do no explicitly state this, so it remains a potential in this categorization.

4.5 Sabotage / Re-enacting Failures

The approaches of sabotage and re-enacting project failures, described in Sections 2.1.7 and 2.1.8, respectively, have a close and obvious tie to the Learning through Failure theory—students are deliberately set up to fail, and this failure is the primary method of learning. In these approaches, students also learn by doing—they learn how to handle project failures, and factors that can cause project failures, by actually experiencing them firsthand. Finally, Situated Learning is also intrinsic to this approach in that the primary characteristic of the learning environment is the presence of realistic adverse events and circumstances.

There are also three other learning theories that this approach has the potential to leverage, if modified slightly. Again, as in several other approaches, Learning through Reflection and Learning through Dialogue could be utilized with dialogue and reflection sessions. These types of activities would be especially useful in reflecting upon failures, and analyzing them to determine their causes, effects, and possible ways to prevent them in the future. Elaboration could also be employed if sabotage "events" (dirty tricks in [47]) or failures are introduced in order of increasing complexity.

4.6 Projects Plus Realism (All Others)

The rest of the "Projects plus Realism" approaches presented in Section 2.1 leverage the same set of learning theories, therefore they are all placed at the end of Table 2. Because the goal of a project in general is to teach students how to perform tasks using the knowledge they are taught, all approaches that involve a project are essentially Learning by Doing approaches. Furthermore, because these approaches all aim to add realism to the environment in which the project is practiced, they also fall into the category of Situated Learning.

Again, any of these approaches could also incorporate reflection and dialogue sessions in order to take advantage of the Learning through Reflection and Learning through Dialogue theories. In addition, most of these techniques could also take advantage of the Elaboration theory in the way they organize the concepts being introduced. For example, one of the approaches that focuses on teaching non-technical skills could teach the simplest non-technical skills first, followed by those that are more advanced.

4.7 Simulation

By far, simulation is the approach that takes advantage of, and has the potential to take advantage of, the largest number of learning theories (eleven). First of all, all aforementioned educational software engineering simulations allow students to learn software processes by participating in them (Learning by Doing), albeit virtually. These simulations also employ Situated Learning by adding realism to the learning environment, although in different ways: Industrial simulations add realistic factors in the form of real project data in the simulation model; Game-based simulations add realism by immersing the student in the role of a participant in a realistic game scenario; Group process simulations inject realism through the simulated characters that behave similarly to real-world participants.

Simulation approaches also strongly fit with the Keller's ARCS model of learning. In particular, they promote attention, relevance, confidence, and satisfaction in the following ways:

- **Attention:** A number of studies done with educational software engineering simulations have repeatedly shown that students find these simulation enjoyable, engaging, and an interesting challenge they are happy to take on [18, 46, 95, 117, 125]. This is particularly true for game-based simulations. Clearly this is the result of the elements of surprise, humor, challenge, and fun that are integral to many game-based simulations.
- **Relevance:** Because learners can experience firsthand how the knowledge they are learning is relevant in a real-world situation (the one that is portrayed in the simulation), simulation promotes a feeling of relevance to students' future careers. This relevance can be enhanced by the usage of real-world data in the model to make the simulation more realistic. Furthermore, as the theory suggests, relevance is enhanced even further if the educational approach builds on previous and present knowledge. Simulations that are used to demonstrate concepts that have already been communicated to the students in another form (e.g., lecture or text) directly address this.
- **Confidence:** Simulations provide a non-trivial challenge that is also doable. As students are given the opportunity to succeed at a simulation, they will feel a

sense of personal confidence in the learning material. This is especially true in game-based simulations, in which students have the additional benefit of feeling they have "won the game."

- **Satisfaction:** As students are able to practice their knowledge and skills in a realistic (yet simulated) setting, seeing the positive consequences of applying their knowledge correctly promotes a true feeling of satisfaction. Again, game-based simulations add to this if the student is also rewarded with a high score or some other game-relevant measure of success.

The exploratory quality of simulation in and of itself also directly implements the Discovery Learning theory. The nature of simulation is highly conducive to allowing students to discover knowledge on their own, as they see phenomena played out in a simulation, and are encouraged to explore, experiment, do research, ask questions, and seek answers.

This type of exploratory learning is also inherently related to the Learning through Failure theory. As students explore the simulation and try different approaches, they are likely to fail at least a few times. In fact, one of the basic purposes of simulations is to allow students to "push boundaries", try different approaches, and fail without fear of the drastic and severe consequences that would occur in a real-world setting. For example, a student who fails in a simulated software project would only have to worry about getting a low game score or seeing an unhappy simulated customer, while in the real world such a failure could cost millions of dollars or have even more serious consequences.

Both Learning through Dialogue and Learning through Reflection have also been incorporated into the simulation approach. However, these have only been used limitedly: with the game-based simulation SESAM [49], and the industrial simulation described in [96]. As mentioned previously, dialogue and reflection sessions have been incorporated into these learning processes as post-simulation activities. In these sessions, students analyze and discuss their simulation experience with a tutor or instructor, and reflect on what they have learned. Some dialogue activity is also an inherent part of Problems and Programmers [18], the educational software engineering card game simulation. Although the dialogue referred to in the Learning through Dialogue theory is usually between a student and teacher, the face-to-face, competitive nature of this physical card game has been shown to promote rich and useful discussion between student opponents, regarding such topics as why they took the approach they did, the reasons behind one person's win and another's loss, and their reactions to unexpected events. This type of student-student dialogue more closely fits within the more general Conversation Theory.

Like Learning through Dialogue and Learning through Reflection, the Elaboration theory has also been only limitedly incorporated into simulation-based software engineering educational approaches. In particular, Elaboration has only been leveraged in the process used with the industrial simulation described in [37]. This process consists of assigning students very simple simulations to begin with, and incrementally increasing the complexity of the simulations as the students progress in their knowledge.

The Lateral Thinking theory is also one that has only been utilized minimally in simulation, but has a strong potential to be used much more, especially in game-based and group process simulations. As described in Section 3.12, the theory of Lateral Thinking states that students will learn best when they are encouraged to look at problems in new and unique ways, within the context of situations that involve events not likely to occur

normally. Simulation technology has clear capability to fit this bill. Simulations, especially those that are computer-based, could be modified in such a way as to allow the student to take on different roles, or the perspectives of different participants in the simulated process. (The inspection group process simulation described in [125] is the only one that has this capability.) Such a feature strongly promotes Lateral Thinking, as it allows a student to look at a software process from the viewpoint of, for instance, a programmer, manager, tester, and customer, all within the same simulation. Furthermore, Lateral Thinking could also be promoted if configurable simulations, such as SESAM and SimSE, are programmed with simulation models that involve the type of "unusual" events proposed by Lateral Thinking.

Although Aptitude-Treatment Interaction has not yet been used in conjunction with educational software engineering simulations, the usage of such simulations could potentially be modified to take advantage of this learning theory. In particular, students with lower aptitudes could be assigned simulation models or activities that are more structured and provide fewer options, while those with high aptitudes are given more open-ended simulations with extra freedom. For instance, a simulation that is trying to teach the waterfall process of software engineering (requirements -> design -> implementation -> test) could have two variations: For lower aptitude students, it could enforce this sequence and never even give them the option of, for example, starting design before requirements are finished. In this way, they would have a very structured experience in which the lessons are obvious and explicit. Higher aptitude students, on the other hand, could be given complete freedom to perform any step at any given time, and would instead have to infer the proper sequence by trying many different approaches and seeing the results of each. Again, this approach might raise some ethical, instructional, and feasibility issues, but still may be worth investigating.

Similarly, incorporating the theory of Multiple Intelligences into simulation in software engineering education may be another avenue worth pursuing. As in the Aptitude-Treatment Interaction theory, different simulations could be given to different students, but based on their learning modalities rather than their aptitudes. It is unclear how or if exactly a simulation could be designed for a student's particular learning modality, and whether this is even a feasible task. Hence, this is designated as an open research area.

Finally, Anchored Instruction is another theory that has not been utilized at all in simulation, but has obvious potential to. In particular, simulations could be used as the anchor (realistic situation, case study, *and* problem, simultaneously) that instruction is centered around. In such a case, students would practice a simulation (or series of simulations) for each concept (or set of concepts) being taught. Simulations would allow for ample exploration—one of the basic tenets of Anchored Instruction—as students could practice the same simulation multiple times, using a different approach each time, learning the consequences of various actions, and, as a result, learning a great deal about the process and concepts being simulated.

4.8 Adding the "Missing Piece"

Although the various "Missing Piece" approaches vary by the particular subject(s) being taught and exact manner of teaching, most of them employ a Learning by Doing approach. Specifically, the subject in question is either incorporated into the class project (e.g., students should learn real-time software engineering, so the software they build for

the project must be real-time), or the student is required to complete a lab assignment or small exercise demonstrating the concept (e.g., have students create a Z specification of an existing piece of software in order to teach them formal methods).

Because the "Missing Piece" is somewhat of a meta-approach, in that it specifies what subject(s) should be added to the curriculum as a whole, rather than how to specifically teach software engineering, the other learning theories are not particularly relevant to this approach on its own.

5. Observations and Recommendations

Table 3 presents the frequency of each software engineering educational approach presented in this paper, including a breakdown of each approach's subcategories. Looking at the number of approaches that fall into the "Projects Plus Realism" category (53 out of 109 total), it is obvious that this is the most popular approach to addressing the problem of adequately preparing students for their future careers in software engineering. This makes logical sense: If we want to better prepare students for the real world, it is only common sense that we try to introduce the real world to them sooner rather than later. However, if we look again at Table 2, we can see that the majority of these approaches only employ two learning theories: Learning by Doing and Situated Learning.

Table 3: Frequency and Breakdown of Each Software Engineering Educational Approach.

Realism	53	Simulation	8	Missing Piece	48
Industrial Partnerships	**16**	**Industrial**	**2**	**Formality**	**3**
- Modify real software	1	**Game-Based**	**4**	- Formal methods	2
- Industrial advisor	1	**Group Process**	**2**	- Engineering	1
- Industrial mentor/lecturer	2			**Process (Specific)**	**21**
- Case study	5			- PSP	14
- Real project / customer	7			- TSP	2
Maintenance/Evolution	**9**			- RUP	3
- Multi-semester	4			- XP	2
- Single-semester	5			**Process (General)**	**6**
Team Composition	**13**			- Process engineering	3
- Long-term teams	1			- Project management	3
- Large teams	3			**Parts of Process**	**3**
- Different C.S. classes	1			- Scenario-based req. eng.	1
- Different majors	2			- Code reviews	1
- Different universities	2			- Usability testing	1
- Different countries	1			**Types of Software Eng.**	**8**
- Team structure	3			- Maintenance/Evolution	3
Non-Technical Skills	**2**			- Component-based SE	2
Open-Endedness	**7**			- Real-time SE	3
- Requirements	2			**Non-Technical Skills**	**7**
- Process	5			- Social/logistical skills	3
Practice-Driven	**3**			- Interact w/ stakeholders	1
Sabotage	**2**			- HCI	2
Project Failures	**1**			- Business aspects	1

In fact, Learning by Doing and Situated Learning are the most heavily-utilized theories in the surveyed approaches. It is obvious that software engineering educators have

gotten very good at employing these two theories in their approaches. However, as a tradeoff, there are many other learning theories that are overlooked. Figure 5 presents the number of approaches that leverage or may potentially leverage each learning theory (approaches that only minimally leverage a theory fall into the potential category in this diagram). From this graph it is clear that there are a number of under-utilized theories. Over half of the learning theories are addressed by less than 50% of the surveyed approaches. Aptitude-Treatment Interaction, Lateral Thinking, and Anchored Instruction are the least utilized theories (aside from Multiple Intelligences, whose potential is unclear in the domain of software engineering education, as discussed in Sections 4.4 and 4.7). Anchored Instruction is employed by only one approach (practice-driven), with the potential to be utilized by only three other approaches (the three simulation categories). Both Lateral Thinking and Aptitude-Treatment Interaction are only potentially or partially employed by four approaches. Clearly, both Learning through Dialogue and Learning through Reflection are the theories with the most potential for greater use—every approach except for the Missing Piece meta-approach could potentially be enhanced to leverage (or better leverage) these theories. Elaboration also has considerable potential to be employed more in these existing approaches (20 of 24).

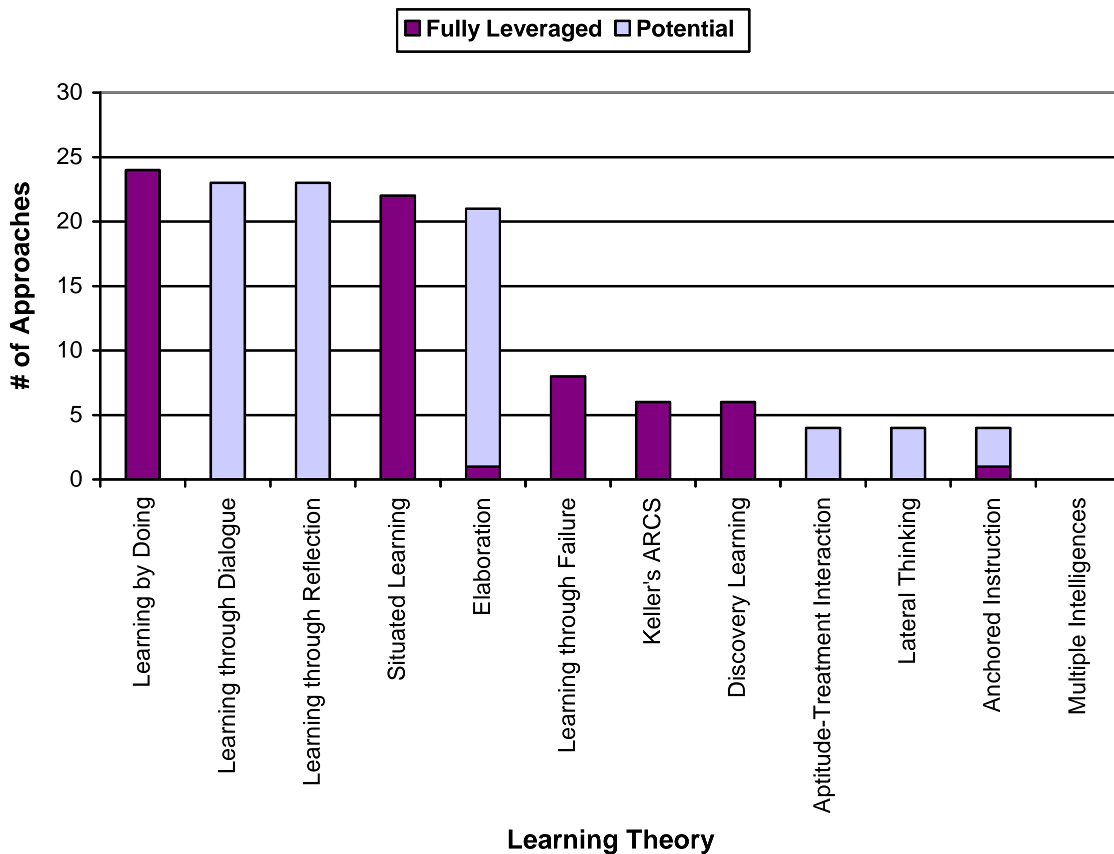

Figure 5: Number of Approaches Leveraging or Potentially Leveraging Each Learning Theory.

If we take a different view and look at each approach rather than each learning theory, we can gain even more insight. Figure 6 depicts the number of learning theories leveraged or potentially leveraged by each approach. (As in Figure 5, theories that are only minimally leveraged are put into the "potential" category.) If we examine each column in the graph, it is evident that most of these approaches are not being maximized in terms of their educational potential. In fact, with the exception of the open-ended approach, the industrial partnership / real project approach, and the missing piece approach, none of them are maximized by more than 50%. Obviously there exists a great deal of untapped potential in each of these approaches.

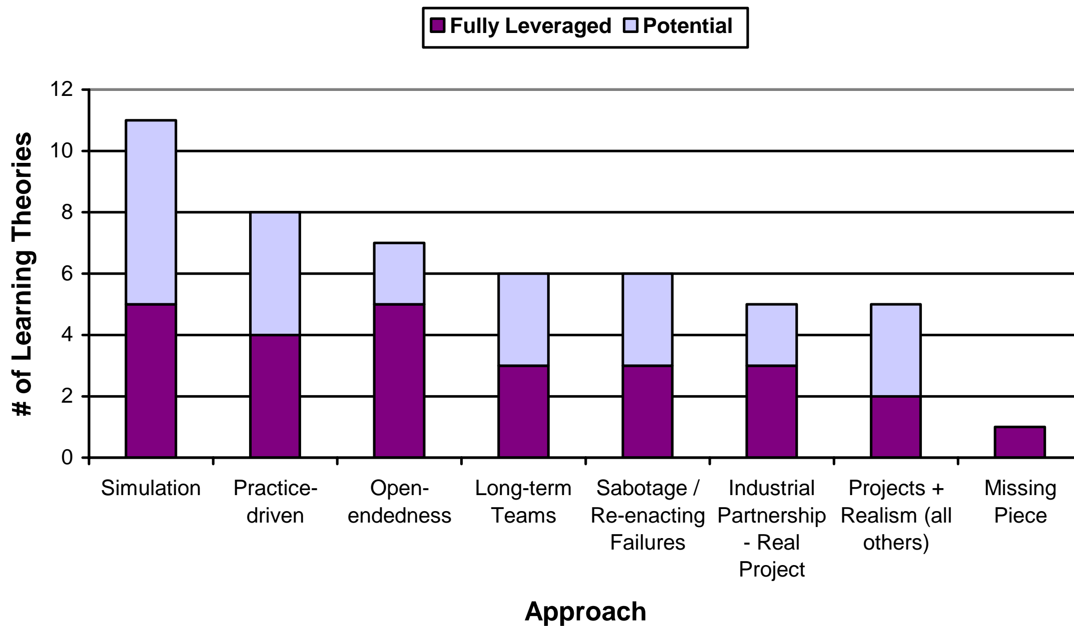

Figure 6: Number of Learning Theories Leveraged or Potentially Leveraged by Each Approach.

Based on this information, I would make the following overall recommendations for modifications (not overhauls) to existing software engineering curricula:

- **The existing approaches should be modified (according to the recommendations presented in Section 4) to take advantage of all these potential areas for improvement.** The fact that nearly every approach only leverages at most 50% of the learning theories it *could* employ reveals that there is a great deal of potential for them to be significantly more educationally effective. Particularly promising are the Learning through Dialogue and Learning through Reflection enhancements, which can be used to enhance nearly every approach. Simulation is the approach that could be improved to employ the most additional learning theories (six). Of course, it is not feasible for one to enhance and incorporate all approaches—a great deal of selectivity and prioritization must be used in choosing which ones are most appropriate to the nature of a specific educational environment. In addition, the previously stated caveat (multiple learning theories must be incorporated naturally and coherently) must be kept in mind.

- **New approaches should be designed to specifically address the more under-utilized learning theories**. Specifically, the least utilized learning theories are Aptitude-Treatment Interaction, Lateral Thinking, and Anchored Instruction. In addition, Learning through Failure, Keller's ARCS, and Discovery Learning are also poorly utilized, being leveraged by less than half of the approaches. It is possible that the reason why these theories are not utilized is because they have little applicability to the domain of software engineering education. Therefore, investigations should also be conducted that determine the true potential of these theories in this domain. Furthermore, studies should be undertaken to assess the relevance of the Multiple Intelligences theory to software engineering education, since it could also potentially be useful to the domain, but its applicability is highly unclear at this point.

Now suppose that, rather than modifying an existing curriculum, we were tasked with designing a brand new software engineering curriculum from the ground up, using only the currently existing approaches (with allowance for modifications to maximize their educational potential). This requires a comparison of these approaches based on the following criteria: Logically, the more educational theories that a particular approach employs, the better and more effective it should be.[5] Therefore, on first glance, one might propose that this ideal curriculum should include all or many of these approaches in a combined fashion in order to maximize the learning benefits. However, such an approach would not only be problematic for the reasons mentioned, but it would also be highly infeasible, due to the fact that both courses and curricula are constrained in terms of time, scope, and resources. While each and every one of these approaches undoubtedly have their merits, and could perhaps be included in an imaginary software engineering curriculum that had no time or resource limits, there is no possible way anyone can incorporate all of them into one course or even one four-year curriculum.

Instead, it is more feasible and efficient to choose those approaches that encompass the most learning theories in and of themselves, and require the minimum amount of time and resources. These approaches could then be combined into a curriculum that makes the optimal tradeoff between feasibility and educational effectiveness. If we look again at Figure 6, we can see that simulation is the clear front-runner in the number of learning theories it addresses (11 of 12). The practice-driven approach (eight of 12) and the open-ended approach (seven of 12) are both relatively close behind, and the remaining approaches (potentially) make use of a gradually decreasing number of theories.

First, let us take a look at the latter two approaches: the practice-driven curriculum and the open-ended approach. While the practice-driven approach may be one of the most educationally rewarding and effective approaches, it also requires perhaps the most radical departure from the traditional lecture-based approach to software engineering education. Furthermore, it is nearly impossible to incorporate it into an existing curriculum that is primarily lecture-based. Instead, it would require a great amount of effort put into a complete redesign of such a curriculum. Perhaps these obstacles are the very reasons behind the low usage of this approach to date (three of 109 total approaches). Very likely,

[5] It should be noted that I am speaking here of singular approaches, not entire curricula. A curriculum that is packed full of all sorts of different approaches that employ a number of different learning theories each would likely be unfocused, overwhelming, and confusing to the student. Rather, combining a few approaches that employ a maximal number of theories would be a more optimal balance.

educators are just not convinced enough of this approach's benefits to warrant putting the effort and risk into employing it. To quell these educators' fears (or else prove them valid), I would suggest the following:

- **Conduct more formal evaluations that demonstrate the effectiveness of the practice-driven approach, especially compared to traditional lecture-based techniques.** To date, only one of these approaches ([97]) has done any such evaluation—they conducted a follow-up study with the graduates from this practice-driven curriculum, which found that their employers reported a decreased need for training them as compared to students who had not been educated using a practice-driven curriculum. This is clearly a promising result. With replication and other types of comparisons, more educators will likely be encouraged to adopt this approach.
- **Accompany practice-driven approaches with reflection & dialogue techniques.** As mentioned previously, the fact that this approach relies on inferred and experiential knowledge rather than explicit knowledge being communicated verbally makes these kinds of activities especially helpful in these situations. Only one of these approaches [129] to date has incorporated reflection and dialogue. Incorporating them more will likely make the approach more effective and result in stronger evidence of its effectiveness when evaluations are conducted.

The potential benefits of a practice-driven curricula have also been recognized by Medhi Jazayeri. He proposes that software engineering curricula should be project-based, in order to "enable the students to apply system-level thinking, see technologies in use, and appreciate the difficulties and benefits of working with others in a team" [70].

Like the practice-driven approach, the open-ended approach is also a technique that is both potentially powerful and potentially risky. Although it requires a less drastic paradigm shift than the practice-driven approach, it is still not widely used (seven of 109 approaches). This is most likely due to the amount of risk and uncertainty that is involved in such an approach. There are undoubtedly a number of unanswered questions in an instructor's mind going into such a project: Will they succeed? Will they fail? If they fail, will they still learn valuable lessons? How do I know that their failures will not be so miserable that they outweigh their successes, or any lessons they may learn? There are three possible ways to address this:

- **Use the open-ended approach only with advanced students, namely those that are in their final year or semester in the program.** This solution is suggested in [94], which argues that only advanced students will have the maturity and sufficient background knowledge to make this a worthwhile exercise with a high probability of success.
- **Use a "watered-down" open-ended approach, in which students are given significant freedom, but the instructor intervenes when they begin to go down the wrong path.** Of course, in this case there still exists some risk on the part of the instructor, namely, where is the balance between letting the students learn a lesson and ensuring they do not throw the project completely off track? When is the right time to intervene and when is it more effective to let the events play themselves out? Obviously, due to the "foggy" nature of this approach, there are some highly subjective issues surrounding its use that can most likely only be resolved on an individual basis.

- **Conduct more formal evaluations of the open-ended approach and compare it to more structured techniques.** All of the open-ended approaches to date have only anecdotal results to report. These are promising, but until conclusive studies are performed, it is unlikely that greater use of this approach will occur.
- **Accompany open-ended approaches with reflection & dialogue sessions.** Like the practice-driven approach, the open-ended approach also relies heavily on knowledge being inferred and built from experience rather than through explicit communication, making this an ideal environment for reflection and dialogue to be helpful. Using these techniques will likely make the approach more effective, which will reflect favorably upon the results of the formal evaluations suggested previously.

Now, let us take a look at the front-runner in this comparison, simulation. Besides incorporating the most learning theories, simulation also has two other characteristics that make it ideal for addressing the needs of software engineering education. First, it is relatively easy to add into a curriculum. An instructor can simply pick up an existing simulation and use it in their class. Simulation runs could be assigned as out-of-class exercises. Minimal preparation, class time, and modification to an existing curriculum would be required in such a situation. Second, the configurable nature of simulations gives them the capacity to teach a wealth of different software engineering sub-topics. It is obvious by the large number of "Missing Piece" approaches (48 out of 109) that there is an enormous number of concepts instructors want to bring into software engineering education. Perhaps the only feasible way a significant number of these subjects could be even introduced in the same curriculum is in teaching them through simulation. While it is probably beyond the capabilities of simulation to teach the low-level details of specific subjects, such as Human-Computer Interaction or formal methods, it could at least be used to show how these concepts fit into the software engineering process at a higher level, which is probably the only feasible level at which to incorporate a number of these subjects into a curriculum. For instance, a simulation designed to highlight the importance of Human-Computer Interaction could penalize the student for not performing usability testing, not involving users early in the development cycle, and/or not creating rapid prototypes of a user interface.

Finally, the history of simulation in education also makes it an ideal tool for software engineering education. Simulations have already been used to teach typical software engineering sub-processes, such as code inspections [125] and requirements analysis meetings [96], and have been known to be effective educational tools in other domains, such as hardware design, pilot training, and battlefield training.

Ironically, although simulation encompasses the most learning theories, is intrinsically simple to add to a curriculum, has the potential to teach a wide range of subjects, and has been used extensively in other educational domains, it is also one of the most under-explored approaches in software engineering education, as can be seen in Table 3 (8 out of 109). Aside from the fact that relatively few simulations exist and new ones are not trivial to build, I believe this can be attributed to the fact that simulation approaches to date have not taken full advantage of their educational potential, and have not been proven to be truly effective in this domain. Specifically, based on the under-explored learning theories discussed in Section 4.7, the following improvements must be made to educational software engineering simulations:

- **Simulations should make better use of dialogue and reflection techniques.** In order to better leverage the Learning through Dialogue and Learning through Reflection theories, simulations should more regularly be accompanied by post-simulation sessions in which instructor-student dialogue and reflection are encouraged. Such an activity would give students the opportunity to analyze, think, and talk about the lessons learned in the simulation, ingraining the concepts more deeply into their minds. (As mentioned previously, such an approach has only been used in [49] and [96]). Such an activity is especially important in simulation, due to the fact that its lessons are usually not completely explicit, but are rather implied and inferred through experience, observation, and analysis.
- **Simulations should be used in order of increasing complexity.** With the exception of [37], students are generally given one simulation of significant complexity and asked to tackle it immediately, with no introduction to simpler simulations first. The Elaboration learning theory could be better utilized if either simple models or simple assignments involving a more complex model (only requiring the student to complete part of, or a certain aspect of, the simulated project, for example) are practiced first, followed by increasingly complex ones.
- **Simulations should allow students to take on different roles in the simulated process.** As mentioned in Section 4.7, Lateral Thinking is one of the learning theories that educational software engineering simulations have failed to fully leverage. A simulation that allows students to experience a software process from different perspectives would address this issue. Such a feature would probably be most feasible and appropriate in game-based and group process simulations, in which there are usually multiple visible characters with different roles.
- **Simulation should take a more central role in instruction.** Also pointed out in Section 4.7 was the failure to leverage the Anchored Instruction learning theory in simulation. An ideal situation that uses simulation as an "anchor" and takes full advantage of this theory would be something like the following: In a lecture, whenever the instructor introduces a concept, she has a simulation model that illustrates that concept. She could first go through the simulation, which is projected on the screen for the students to see. Then, the students (who all have laptops) would be assigned to execute the simulation on their own. For instance, in teaching the waterfall model of software development, the instructor could assign them to first follow the correct requirements -> design -> code -> test cycle, then start the simulation again and follow a different model, and compare the results of each simulation run. Of course, this type of approach would require a significant amount of effort on the instructor's part, re-designing their teaching approach and building models for each concept, but it also may be the most rewarding and effective way to incorporate simulation into a course (as opposed to, e.g., simply assigning simulations as homework).
- **Simulations should be more easily configurable.** If simulations are going to take a more central role in instruction and teach a wide range of different concepts (necessitated by the "Missing Piece" approaches), this obviously requires that they be easily configurable. Such configurability could be realized through two major features: user-friendly modeling tools that allow instructors to build models quickly and easily; and pre-existing, possibly generic models that can be easily

modified, enhanced, and/or configured to fit different educational goals. The current state-of-the-art in configurability is SESAM, which has its own complicated (although powerful) textual process modeling language that the instructor must learn. Furthermore, there has only been one SESAM model built to date [49]. This does not give an instructor many examples to work with when trying to build a new model, and is also evidence that, despite SESAM's powerful language, the need to actually textually program a model is a significant challenge that few wish to tackle.

- **Simulation approaches should be more robustly verified.** Possibly one of the most significant factors in the underutilization of simulation in software engineering education is that instructors are unsure of its effectiveness in this domain, and therefore hesitant to expend the time and effort to invest in employing it in their classrooms. This is a valid concern, as there have been relatively few studies that have definitively affirmed the effectiveness of simulation in software engineering education. With the exception of [100], all of the other experiments involving educational software engineering simulations, although mostly favorable (a possible explanation for the unfavorable results of [49] is that it did not follow all of these recommendations), have been preliminary and informal in nature. More robust, complete, and formally-design experiments are needed to confirm these studies and/or reveal needed areas of improvement. Such studies will likely instill more confidence in the minds of software engineering educators who are considering adding simulation to their curriculum.

In order to achieve these goals, it is unclear whether existing simulations can be modified, or whether this will require new simulations to be developed. It is also unclear whether it is even feasible to have all of these features in one simulation. For instance, a simulation that allows different viewpoints might have the adverse side effect of making configurability quite difficult. In order to answer these questions, a series of feasibility studies need to be undertaken: First, existing simulations must be explored in order to see if they can be modified to fit these recommendations. Second, either an existing simulation should be modified, or a new simulation be created that attempts to incorporate all of these features and explores the tradeoffs between them. Only after these questions are answered can the ideal and most educationally effective software engineering simulation environment (and associated approach for class use) be developed.

From this discussion, it should be clear that although simulation has tremendous educational potential, its effectiveness is highly contingent on the method of use in a curriculum, and the "right" usage will require a significant amount of effort—it is not a ready-made, pre-packaged "silver bullet." Specifically, there are several learning theories that can only be leveraged if simulation is used in a particular way. Anchored Instruction, for instance, is only leveraged if numerous realistic simulation models are built and the curriculum is redesigned around these examples. Even in a non-Anchored Instruction approach in which lectures are the main mode of teaching and simulation is used as a supporting activity, simulation models must be built to illustrate the concepts the instructor wishes to convey. Likewise, Aptitude-Treatment Interaction and Elaboration can only be employed if multiple models of varying levels of structure and complexity are built. Aptitude-Treatment Interaction also requires the further effort of assessing students' aptitudes and assigning models accordingly. Finally, Learning through Dialogue and Learning

through Reflection can only be incorporated if the curriculum is modified to include sessions for these purposes. One should not think that simply adopting simulation in a minimal way, such as adding it as a one-time homework assignment, is going to bring the maximal educational benefits, although it may bring some advantages.

Although this is the first time simulation has emerged as one of the answers to the problem of software engineering education through the particular analysis presented in this paper, it is certainly not the first time simulation in software engineering education has been proposed as a highly promising approach. In Mary Shaw's Roadmap for Software Engineering Education, one of the key improvements she suggests is to "exploit our own technology in support of education… In local classrooms, we could make better use of simulations and game-playing exercises" [118]. At Ed Yourdon's keynote speech at the 2002 Conference on Software Engineering Education and Training titled "Preparing Software Engineers for the 'Real World'", he stated that industry has found that the best way to teach their new hires (recent graduates) the competencies they lack is through "'war games' based on realistic simulation models of software projects", and suggested that academia also incorporate this kind of approach into their curricula [141]. A project that aimed to analyze "the gap between the knowledge and skills learned in academic software engineering course projects and those required in real projects" by assessing a typical software engineering curriculum using the guidelines presented in SWEBOK concluded that the gaps found are attributable to the time and size constraints on course projects. They suggested that the ideal way to bridge this gap would be through "simulation technology to create a large project learning environment" [82]. Clearly, there are many experts in the field that are in agreement on the potential benefits of simulation in software engineering education.

6. Related Work

Because software engineering education has long been recognized as being problematic, there have been several other papers written that provide general, overall suggestions for remedying its problems. Most of these have gone about suggesting solutions either through intuition, informal personal analysis, or the author's experience and expertise. Mentioned previously, Medhi Jazayeri gave a keynote talk at the 2004 Automated Software Engineering Conference that gave a broad outline of his ideal software engineering curriculum, largely based on new trends and technologies in the field, arguing that these should be major driving factors in the design of software engineering educational programs [70]. Bertrand Meyer takes a different approach in his 2001 IEEE Computer article about the topic, stating that software engineering curricula should instead be more rooted in traditional mathematical, engineering, and computer science topics and techniques [91]. Mary Shaw's roadmap for software engineering education took a stance somewhere in between these two viewpoints, stating that an engineering attitude should be instilled in students while, at the same time, education should be kept current in the face of rapid change [118]. She also recommended that distinct roles in software engineering should be defined, and separate education programs be developed for each.

While these analyses are all based on the expertise and experience of seasoned software engineers and educators, there is also another class of analyses that are based on data other than personal experience. Also mentioned previously is an analysis of the deficiencies of software engineering education based on the areas of knowledge defined in

SWEBOK [82]. This study concluded that the primary problem is the time and scope constraints of the academic environment preventing students from being able to practice projects of realistic size, and suggested simulation as a solution. In a 1987 paper, when software engineering education was still an up-and-coming field, Doris Carver surveyed software engineering textbooks to glean what the important topics in the area were, and accordingly proposed an ideal curriculum for a software engineering course [34]. Finally, Timothy Lethbridge conducted a survey of computer science or software engineering graduates working in industry, asking them which topics they felt turned out to be most useful to them in their careers [80]. From this, he recommended which topics should be taught more (such as data structures, algorithms, communication, and professionalism) based on their higher anticipated usefulness, as well as ones that should be taught less (such as mathematics). In a 1997 article in IEEE Software, Thomas Hilburn makes recommendations for improving software engineering education based on Carnegie Mellon's Software Engineering Institute's people, process, and technology triad. Namely, he defines a set of skills in each category (i.e., people skills, technology skills, and process skills) that education should focus on and invest in [66].

This survey is distinguished from the others in three major ways: First, it is the only one to date that approaches the problem from a general educational view, using general learning theories, rather than from a domain-specific software engineering education view. Second, this is the first time an extensive survey has been done of the research literature in the field. Third, most of these other works have focused primarily on topics that should be included in software engineering educational programs. Although topics are a necessary part of any discussion about software engineering education, including this survey, and some delivery methods are largely defined by the topics they are meant to teach (e.g., the various team composition approaches are meant to teach the topic of teamwork skills, the missing piece approach is defined by its topics), the main focus here is delivery methods and overall approaches to teaching the subject as a whole.

7. Conclusions

Despite all of the effort that educators have put into making software engineering educational techniques more effective, new graduates are still notoriously unprepared for functioning in a real-world software engineering situation. In this paper, I have attempted to discover one reason why this is so. An analysis of software engineering educational delivery methods in light of learning theories revealed three possible answers to this:

1. There are several learning theories that are under-utilized in existing approaches. In particular, Learning through Failure, Keller's ARCS, Discovery Learning, Aptitude-Treatment Interaction, Lateral Thinking, and Anchored Instruction are all leveraged by less than half of the approaches. Learning by Doing and Situated Learning are the most utilized theories.
2. There exists a great deal of untapped potential in the existing approaches that suggests ways they could be enhanced to take advantage of several more learning theories.
3. The approaches that utilize the widest range of learning theories and are therefore some of the most promising, namely, practice-driven curricula, the open-ended approach, and simulation, are infrequently used. This could be attributed to certain shortfalls in the existing approaches in these categories and their typical us-

age. Specifically, practice-driven curricula is a more radical departure from traditional types of instruction than most educators are comfortable with; open-ended approaches entail a great deal of risk on the part of the instructor; and existing simulations fall short in their usage, flexibility, and configurability.

Based on these holes and deficiencies, the following recommendations can be made for enhancing the educational effectiveness of software engineering curricula:

1. **Existing software engineering educational approaches should be modified to take advantage of stated potential areas for improvement.** Most promising are the Learning through Dialogue and Learning through Reflection enhancements, which can be used to enhance nearly every approach. Simulation is the approach that could be improved to employ the most additional learning theories (six).
2. **New approaches should be designed to specifically address the more under-utilized learning theories**. Over half of the learning theories are utilized by 50% or less of the approaches. Most under-utilized are Aptitude-Treatment Interaction, Lateral Thinking, and Anchored Instruction. Although the theory of Multiple Intelligences is not utilized by any approach, it also does not have clear potential to be. Hence, there are needed investigations into this theory's applicability to the domain of software engineering education.
3. **Practice-driven approaches should be more robustly evaluated and make better use of reflection and dialogue techniques.** With more conclusive evidence that this approach is effective, along with the added benefits of reflection and dialogue techniques, perhaps more educators will be encouraged to adopt this somewhat radical approach.
4. **Open-ended approaches should be used with only advanced students, perhaps be tempered with a slightly more structured approach, make use of reflection and dialogue techniques, and be more robustly verified.** The combination of these modifications will likely make the approach more effective and viewed as less risky by potential adopters of it.
5. **Simulations should be accompanied by dialogue and reflection sessions, used in order of increasing complexity, allow students to take on different roles and viewpoints, take a more central role in instruction, be more easily configurable, and undergo more robust evaluation.** These modifications will directly address what are the probable factors that have impeded the widespread use of simulation in the past, and make it a much more potentially educationally effective technology.
6. **Software engineering educational techniques should undergo more frequent, robust, and formal evaluations.** While there exists a wealth of promising approaches, many of them are risky and require a significant paradigm shift on the part of the educator, as well as a great deal of effort incorporating them into their curricula. If these techniques are proven to be effective, more educators will likely be willing to expend the effort to adopt them.
7. **Software engineering educators should become more educated in learning theories.** If educators are better versed in learning theories, they will design their delivery methods to better take advantage of these theories.
8. **Software engineering education research should be framed in the context of learning theories.** Through this perspective, the community can begin to ap-

proach the problem of software engineering education's effectiveness from a new and potentially useful angle.

To sum up, software engineering education could become more effective if delivery methods are designed, modified, and evaluated with learning theories in mind

The particular analysis presented in this paper, evaluating software engineering educational approaches in terms of the learning theories they leverage, is, of course, only one analysis, and surely does not reveal all of the problems and holes and suggest all of the potential solutions. There are undoubtedly numerous other ways and criteria against which software engineering educational approaches can be evaluated that may reveal several different insights. However, the suggestions presented in this paper provide a good starting point for improving the state of software engineering based on this particular analysis.

Acknowledgements

I would like to thank my advisor, André van der Hoek, for his invaluable wisdom, guidance, and feedback on this paper.

References

1. *The Journal of Systems and Software*. Vol. 49. 1999: Elsevier Science Inc.
2. *Proceedings of the 22nd International Conference on Software Engineering*. 2000: ACM.
3. *Proceedings of the 23rd International Conference on Software Engineering*. 2001, Toronto, Canada: IEEE Computer Society.
4. *IEEE Software, Special Issue on Educating Software Professionals*. Vol. 19 (5), September/October. 2002: IEEE.
5. *Proceedings of the 24th International Conference on Software Engineering*. 2002, Orlando, Florida: ACM.
6. *Proceedings of the 2003 Frontiers in Education Conference*. 2003, Boulder, CO, USA.
7. *Proceedings of the 2004 Canadian Conference on Computer and Software Engineering Education*. 2004, Calgary, Alberta, Canada.
8. *Proceedings of the 2004 Frontiers in Education Conference*. 2004, Savannah, GA, USA.
9. *Proceedings of the Seventeenth Conference on Software Engineering Education and Training*. 2004, Norfolk, Virginia, USA: IEEE.
10. Abernethy, K. and J. Kelly, *Technology Transfer Issues for Formal Methods of Software Specification*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 23-31.
11. Ace Win-E-Crypt 1.0 Beta. 2001.
12. Alessi, S.M. and S.R. Trollip, *Multimedia for Learning*. 2001, Needham Heights, MA, USA: Allyn & Bacon.
13. Alfonso, M.I. and F. Mora, *Learning Software Engineering with Group Work*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 309-316.

14. Anderson, R.C., *The Notion of Schemata and the Educational Enterprise*, in *Schooling and the Acquisition of Knowledge*, R.C. Anderson, R.J. Spiro, and W.E. Monatague, Editors. 1977, Lawrence Erlbaum: Hillsdale, NJ, USA. p. 415-431.
15. Andrews, J.H. and H.L. Lutfiyya, *Experience Report: A Software Maintenance Project Course*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 132-139.
16. Angehrn, A.A., *Advanced Social Simulations: Innovating the Way we Learn how to Manage Change in Organizations.* International Journal of Information Technology Education, 2004 (to appear).
17. Applefield, J.M., R.L. Huber, and M. Moallem, *Constructivism in Theory and Practice Toward a Better Understanding.* High School Journal, 2000. **84**(2): p. 35-46.
18. Baker, A., E.O. Navarro, and A. van der Hoek, *Problems and Programmers: An Educational Software Engineering Card Game*, in *Proceedings of the 2003 International Conference on Software Engineering*. 2003: Portland, Oregon. p. 614-619.
19. Balci, O., *Annals of Software Engineering*. Vol. 6. 1998: Baltzer Science Publishers.
20. Beckman, K., et al., *Collaborations: Closing the Industry-Academia Gap.* IEEE Software, 1997. **14**(6): p. 49-57.
21. Bernstein, L. and D. Klappholz, *Eliminating Aversion to Software Process in Computer Science Students and Measuring the Results*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY, USA. p. 90-99.
22. Blake, B.M., *A Student-Enacted Simulation Approach to Software Engineering Education.* IEEE Transactions on Education, 2003. **46**(1): p. 124-132.
23. Blake, B.M. and T. Cornett, *Teaching an Object-Oriented Software Development Lifecycle in Undergraduate Software Engineering Education*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY. p. 234-240.
24. Bonwell, C.C. and J.A. Eison, *Active Learning: Creating Excitement in the Classroom.* ERIC Clearinghouse on Higher Education, 1991(Document No. ED 340 272).
25. Bransford, J.D., et al., *Anchored Instruction: Why we Need it and how Technology can Help*, in *Cognition, Education, and Multimedia: Exploring Ideas in High Technology*, D. Nix and R. Spiro, Editors. 1990, Lawrence Erlbaum: Hillsdale, NJ. p. 115-141.
26. Brereton, O.P., M. Gumbley, and S. Lees, *Distributed Student Projects in Software Engineering*, in *Proceedings of the Eleventh Conference on Software Engineering Education and Training*. 1998, IEEE: Atlanta, GA, USA. p. 4-15.
27. Brereton, O.P., et al., *Student Group Working Across Universities: A Case Study in Software Engineering.* IEEE Transactions on Education, 2000. **43**(4): p. 394-399.
28. Brown, J.S., A. Collins, and P. Duguid, *Situated Cognition and the Culture of Learning.* Educational Researcher, 1989. **18**(1): p. 32-42.

29. Brown, S.M., *A Software Maintenance Process Architecture*, in *Proceedings of the Ninth Conference on Software Engineering Education and Training*. 1996, IEEE: Daytona Beach, FL, USA. p. 130-141.
30. Bruner, J., *Acts of Meaning*. 1990, Cambridge, MA, USA: Harvard University Press.
31. Callahan, D. and B. Pedigo, *Educating Experienced IT Professionals by Addressing Industry's Needs.* IEEE Software, 2002. **19**(5): p. 57-62.
32. Cannon, B., T. Hilburn, and J. Diaz-Herrera, *Tutorial: Introduction to the Team Software Process*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, 315: Austin, TX, USA.
33. Carrington, D., B. McEniery, and D. Johnston, *PSP in the Large Class*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 81-88.
34. Carver, D.L., *Recommendations for Software Engineering Education*, in *Proceedings of the Eighteenth SIGCSE Technical Symposium on Computer Science Education*. 1987: St. Louis, MO, USA. p. 228-232.
35. Ceberio-Verghese, A.C., *Personal Software Process: A User's Perspective*, in *Proceedings of the Ninth Conference on Software Engineering Education and Training*. 1996, IEEE: Daytona Beach, FL, USA. p. 52-65.
36. Collins, A., *Cognitive Apprenticeship and Instructional Technology*, in *Educational Values and Cognitive Instruction: Implications for Reform*, L. Idol and B.F. Jones, Editors. 1991, Erlbaum: Hillsdale, NJ.
37. Collofello, J.S., *University/Industry Collaboration in Developing a Simulation Based Software Project Management Training Course*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*, S. Mengel and P.J. Knoke, Editors. 2000, IEEE Computer Society. p. 161-168.
38. Conn, R., *Developing Software Engineers at the C-130J Software Factory.* IEEE Software, 2002. **19**(5): p. 25-29.
39. Cook, J., *The Role of Dialogue in Computer-based Learning and Observing Learning: an Evolutionary Approach to Theory.* Journal of Interactive Media in Education, 2002. **5**.
40. Cowling, A.J., *The Crossover Project as an Introduction to Software Engineering*, in *Proceedings of the Seventeenth Conference on Software Engineering Education and Training*. 2004, IEEE: Norfolk, VA, USA. p. 12-17.
41. Crnkovic, I., R. Land, and A. Sjogren, *Is Software Engineering Training Enough for Software Engineers?* in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain.
42. Cronbach, L. and R. Snow, *Aptitudes and Instructional Methods: A Handbook for Research on Interactions*. 1977, New York, NY, USA: Irvington.
43. Dalcher, D. and M. Woodman, *Together We Stand: Group Projects for Integrating Software Engineering in the Curriculum*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain.
44. Dampier, D.A. and R.E. Wilson, *Teaching Scientific Method for Real-Time Software Engineering*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 199-199.

45. Daniels, M., X. Faulkner, and I. Newman, *Open Ended Group Projects, Motivating Students and Preparing them for the 'Real World'*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY, USA. p. 115-126.
46. Dantas, A.R., M.O. Barros, and C.M.L. Werner, *A Simulation-Based Game for Project Management Experiential Learning*, in *Proceedings of the 2004 International Conference on Software Engineering and Knowledge Engineering*. 2004: Banff, Alberta, Canada.
47. Dawson, R., *Twenty Dirty Tricks to Train Software Engineers*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 209-218.
48. DeBono, E., *NewThink: The Use of Lateral Thinking in the Generation of New Ideas*. 1967, New York, NY, USA: Basic Books.
49. Drappa, A. and J. Ludewig, *Simulation in Software Engineering Training*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 199-208.
50. Favela, J. and F. Pena-Mora, *An Experience in Collaborative Software Engineering Education.* IEEE Software, 2001. **18**(2): p. 47-53.
51. Flor, N.V., F.J. Lerch, and S. Hong, *A Market-driven Approach to Teaching Software Components Engineering.* Annals of Software Engineering, 1998. **6**: p. 223-251.
52. Gamble, R.F. and L.A. Davis, *A Framework for Interaction in Software Development Training.* Journal of Information and Technology Education, 2002. **1**(4).
53. Gardner, H., *Art, Mind and Brain.* 1982, New York, NY, USA: Basic Books.
54. Gee, J.P., *What Video Games Have to Teach Us About Literacy and Learning*. 2003, New York, NY, USA: Palgrave Macmillan.
55. Gehrke, M., et al., *Reporting about Industrial Strength Software Engineering Courses for Undergraduates*, in *Proceedings of the 24th International Conference on Software Engineering*. 2002, IEEE: Orlando, FL, USA. p. 395-405.
56. Gnatz, M., et al., *A Practical Approach of Teaching Software Engineering*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 120-128.
57. Goold, A. and P. Horan, *Foundation Software Engineering Practices for Capstone Projects and Beyond*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY, USA. p. 140-146.
58. Groth, D.P. and E.L. Robertson, *It's All About Process: Project-Oriented Teaching of Software Engineering*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 7-17.
59. Habra, N., *"Peopleware" Integration in an Evolving Software Engineering Curriculum*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA. p. 102-110.
60. Halling, M., et al., *Teaching the Unified Process to Undergraduate Students*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY, USA. p. 148-159.

61. Harrison, J.V., *Enhancing Software Development Project Courses Via Industry Participation*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA.
62. Hayes, J.H., *Energizing Software Engineering Education through Real-World Projects as Experimental Studies*, in *Proceedings of the 15th Conference on Software Engineering Education and Training*. 2002, IEEE. p. 192-206.
63. Hazzan, O. and Y. Dubinsky, *Teaching a Software Development Methodology: The Case of Extreme Programming*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 176-184.
64. Hazzan, O. and J.E. Tomayko, *Reflection Processes in the Teaching and and Learning of Human Aspects of Software Engineering*, in *Proceedings of the Seventeenth Conference on Software Engineering Education and Training*. 2004, IEEE: Norfolk, VA, USA. p. 32-38.
65. Hilburn, T., *PSP Metrics in Support of Software Engineering Education*, in *Proceedings of the Twelfth Conference on Software Engineering Education and Training*. 1999, IEEE: New Orleans, LA, USA. p. 135-136.
66. Hilburn, T.B., *Software Engineering Education: A Modest Proposal.* IEEE Software, 1997: p. 44-48.
67. Hirai, K., *Micro-Process Based Software Metrics in the Training*, in *Proceedings of the Twelfth Conference on Software Engineering Education and Training*. 1999, IEEE: New Orleans, LA, USA. p. 132-134.
68. IEEE Computer Society Professional Practices Committee, *Guide to the Software Engineering Body of Knowledge*. 2004.
69. Jaccheri, M.L. and P. Lago, *Applying Software Process Modeling and Improvement in Academic Setting*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA. p. 13-27.
70. Jazayeri, M., *The Education of a Software Engineer*, in *Proceedings of the 19th International Conference on Automated Software Engineering*. 2004, IEEE: Linz, Austria. p. xviii-xxvii.
71. Keller, J.M. and K. Suzuki, *Use of the ARCS Motivation Model in Courseware Design*, in *Instructional Designs for Microcomputer Courseware*, D.H. Jonassen, Editor. 1988, Lawrence Erlbaum: Hillsdale, NJ, USA.
72. Kessler, R.R. and L.A. Williams, *"If This is What It's Really Like, Maybe I Better Major in English": Integrating Realism into a Sophomore Software Engineering Course*, in *Proceedings of the 1999 Frontiers in Education Conference*. 1999, IEEE: San Juan, Puerto Rico.
73. Knowles, M.S., *The Modern Practice of Adult Education From Pedagogy to Andragogy*. 1980, Chicago, IL, USA: Association Press.
74. Kolb, D.A., *Experiential Learning: Experiences as the Source of Learning and Development*. 1984, Englewood Cliffs, NJ, USA: Prentice-Hall International, Inc.
75. Kornecki, A.J., *Real-Time Computing in Software Engineering Education*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 197-198.

76. Kornecki, A.J., I. Hirmanpour, and M. Towhidnajad, *Strengthening Software Engineering Education through Academic Industry Collaboration*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA. p. 204-211.
77. Kornecki, A.J., S. Khajenoori, and D. Gluch, *On a Partnership between Software Industry and Academia*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 60-69.
78. Kornecki, A.J., J. Zalewski, and D. Eyassu, *Learning Real-Time Programming Concepts through VxWorks Lab Experiments*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 294-301.
79. Krone, J., D. Juedes, and M. Sitharam, *When Theory Meets Practice: Enriching the CS Curriculum Through Industrial Case Studies*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY, USA. p. 207-214.
80. Lethbridge, T.C., *Priorities for the Education and Training of Software Engineers.* Journal of Systems and Software, 2000. **53**(1): p. 53-71.
81. Lisack, S.K., *The Personal Software Process in the Classroom: Student Reactions (An Experience Report)*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 169-175.
82. Ludi, S. and J.S. Collofello, *An Analysis of the Gap Between the Knowledge and Skills Learned in Academic Software Engineering Course Projects and Those Required in Real Projects*, in *Proceedings of the 2001 Frontiers in Education Conference*. 2001.
83. Maletic, J.I., A. Howald, and A. Marcus, *Incorporating PSP into a Traditional Software Engineering Course: An Experience Report*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 89-97.
84. Mandl-Striegnitz, P., *How to Successfully Use Software Project Simulation for Educating Software Project Managers*, in *Proceedings of the 2001 Frontiers in Education Conference*. 2001: Reno, NV, USA.
85. Mayr, H., *Teaching Software Engineering by Means of a "Virtual Enterprise"*, in *Proceedings of the 10th Conference on Software Engineering*. 1997, IEEE Computer Society. p. 176-184.
86. McDonald, J., *Teaching Software Project Management in Industrial and Academic Environments*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 151-160.
87. McKeachie, W., *Teaching Tips: Strategies, Research, and Theory for College and University Teachers*. 1994, Lexington, MA, USA: D.C. Heath and Company.
88. McKim, J.C. and H.J.C. Ellis, *Using a Multiple Term Project to Teach Object Oriented Programming and Design*, in *Proceedings of the Seventeenth Conference on Software Engineering Education and Training*. 2004, IEEE: Norfolk, VA, USA.
89. McMillan, W.W. and S. Rajaprabhakaran, *What Leading Practitioners Say Should Be Emphasized in Students' Software Engineering Projects*, in *Proceed-*

ings of the Twelfth Conference on Software Engineering Education and Training, H. Saiedian, Editor. 1999, IEEE Computer Society. p. 177-185.
90. Mengel, S.A., L. Carter, and J. Falkenberg, *A Perspective on Three Cooperating Courses*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 265-272.
91. Meyer, B., *Software Engineering in the Academy.* IEEE Computer, 2001. **34**(5): p. 28-35.
92. Miller, G.A., *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.* Psychological Review, 1956. **63**: p. 81-97.
93. Murphy, M.G., *Teaching Software Project Management: A Response-Interaction Approach*, in *Proceedings of the Twelfth Conference on Software Engineering Education and Training*. 1999, IEEE: New Orleans, LA, USA. p. 26-31.
94. Navarro, E.O. and A. van der Hoek, *Scaling Up: How Thirty-two Students Collaborated and Succeeded in Developing a Prototype Software Design Environment*, in *Proceedings of the Eighteenth Conference on Software Engineering Education and Training*. 2004, IEEE: Ottawa, Canada (to appear).
95. Navarro, E.O. and A. van der Hoek, *Design and Evaluation of an Education Software Process Simulation Environment and Associated Model*, in *Proceedings of the Eighteenth Conference on Software Engineering Education and Training*. 2005 (to appear), IEEE: Ottawa, Canada.
96. Nulden, U. and H. Scheepers, *Understanding and Learning about Escalation: Simulation in Action*, in *Proceedings of the 3rd Process Simulation Modeling Workshop (ProSim 2000)*. 2000: London, United Kingdom.
97. Ohlsson, L. and C. Johansson, *A Practice Driven Approach to Software Engineering Education.* IEEE Transactions on Education, 1995. **38**(3): p. 291-295.
98. Parrish, A., et al., *A Case Study Approach to Teaching Component Based Software Engineering*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 140-147.
99. Pask, G., *Conversation, Cognition, and Learning*. 1975, New York, NY, USA: Elsevier.
100. Pfahl, D., et al., *Evaluating the Learning Effectiveness of Using Simulations in Software Project Management Education: Results From a Twice Replicated Experiment.* Information and Software Technology, 2004. **46**: p. 81-147.
101. Pierce, K.R., *Teaching Software Engineering Principles Using Maintenance-Based Projects*, in *Proceedings of the 10th Conference on Software Engineering Education and Training*. 1997, IEEE Computer Society: Virginia Beach, VA, USA. p. 53-60.
102. Poole, W.G., *The Softer Side of Custom Software Development: Working with the Other Players*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 14-21.
103. Postema, M., J. Miller, and M. Dick, *Including Practical Software Evolution in Software Engineering Education*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 127-135.
104. Prensky, M., *Digital Game-Based Learning*. 2001, New York, NY: McGraw-Hill.

105. Reigeluth, C.M. and C.A. Rodgers, *The Elaboration Theory of Instruction: Prescriptions for Task Analysis and Design.* NSPI Journal, 1980. **19**: p. 16-26.
106. Resnick, L., *Learning in School and Out.* Educational Researcher, 1987. **16**(9): p. 13-20.
107. Robillard, P.N., *Measuring Team Activities in a Process-Oriented Software Engineering Course*, in *Proceedings of the Eleventh Conference on Software Engineering Education and Training*. 1998, IEEE: Atlanta, GA, USA. p. 90-101.
108. Robillard, P.N., P. Kruchten, and P. d'Astous, *YOOPEEDOO (UPEDU): A Process for Teaching Software Process*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 18-26.
109. Rogers, C.R., *Freedom to Learn*. 1969, Columbus, OH, USA: Merrill.
110. Runeson, P., *Experiences from Teaching PSP for Freshmen*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 98-107.
111. Schank, R.C., *Virtual Learning*. 1997, New York, NY, USA: McGraw-Hill.
112. Schank, R.C. and C. Cleary, *Engines for Education*. 1995, Hillsdale, NJ, USA: Lawrence Erlbaum Associates, Inc.
113. Schlimmer, J.C., J.B. Fletcher, and L.A. Hermens, *Team-Oriented Software Practicum.* IEEE Transactions on Education, 1994. **37**(2): p. 212-220.
114. Schlimmer, J.C. and J.R. Hagemeister, *Utilizing Corporate Models in a Software Engineering Studio*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA.
115. Schon, D., *Educating the Reflective Practitioner*. 1987, San Francisco, CA, USA: Jossey-Bass.
116. Sebern, M.J., *The Software Development Laboratory: Incorporating Industrial Practice in an Academic Environment*, in *Proceedings of the 15th Conference on Software Engineering and Training*. 2002, IEEE. p. 118-127.
117. Sharp, H. and P. Hall, *An Interactive Multimedia Software House Simulation for Postgraduate Software Engineers*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 688-691.
118. Shaw, M., *Software Engineering Education: A Roadmap*, in *The Future of Software Engineering*, A. Finkelstein, Editor. 2000, ACM. p. 373-380.
119. Shukla, A. and L. Williams, *Adapting Extreme Programming for a Core Software Engineering Course*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE. p. 184-191.
120. Sikkel, K., T.A.M. Spil, and R.L.W. van de Web, *Replacing a Hospital Information System: an Example of a Real-World Case Study*, in *Proceedings of the Twelfth Conference on Software Engineering Education and Training*. 1999, IEEE: New Orleans, LA, USA.
121. Sindre, G., et al., *The Cross-Course Software Engineering Project at the NTNU: Four Years of Experience*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 251-258.

122. Slimick, J., *An Undergraduate Course in Software Maintenance and Enhancement*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA. p. 61-73.
123. Stepien, W.J. and S.A. Gallagher, *Problem-based Learning: As Authentic as it Gets.* Educational Leadership, 1993. **50**(7): p. 25-28.
124. Sternberg, R.J., R.K. Wagner, and L. Okagaki, *Practical Intelligence: The Nature and Role of Tacit Knowledge in Work and at School*, in *Mechanisms of Everyday Cognition*, J.M. Puckett and H.W. Reese, Editors. 1993, Lawrence Erlbaum Associates: Hillsdale, NJ. p. 205-227.
125. Stevens, S.M., *Intelligent Interactive Video Simulation of a Code Inspection.* Communications of the ACM, 1989. **32**(7): p. 832-843.
126. Sticht, T.G., *Applications of the Audread Model to Reading Evaluation and Instruction*, in *Theory of Practice and Early Reading*, L. Resnick and P. Weaver, Editors. 1975, Erlbaum: Hillsdale, NJ.
127. Suri, D. and M.J. Sebern, *Incorporating Software Process in an Undergraduate Software Engineering Curriculum: Challenges and Rewards*, in *Proceedings of the Seventeenth Conference on Software Engineering Education and Training*. 2004, IEEE: Norfolk, VA, USA. p. 18-23.
128. Syu, I., et al., *A Web-Based System for Automating a Disciplined Personal Software Process (PSP)*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA. p. 86-96.
129. Tomayko, J.E., *Carnegie Mellon's Software Development Studio: a Five Year Retrospective*, in *Proceedings of the Ninth Conference on Software Engineering Education and Training*. 1996, IEEE: Daytona Beach, FL, USA. p. 119-129.
130. Umphress, D.A. and J.A. Hamilton, *Software Process as a Foundation for Teaching, Learning, and Accrediting*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, Kentucky, USA. p. 160-169.
131. Upchurch, R.L. and J.E. Sims-Knight, *Designing Process-based Software Curriculum*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA. p. 28-38.
132. Upchurch, R.L. and J.E. Sims-Knight, *In Support of Student Process Improvement*, in *Proceedings of the Eleventh Conference on Software Engineering Education and Training*. 1998, IEEE: Atlanta, GA, USA. p. 114-123.
133. van der Veer, G. and H. van Vliet, *The Human-Computer Interface is the System; A Plea for a Poor Man's HCI Component in Software Engineering Curricula*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*. 2001, IEEE: Charlotte, NC, USA. p. 276-286.
134. Vaughn, R.B., *A Report on Industrial Transfer of Software Engineering to the Clasroom Environment*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 15-22.
135. von Konsky, B.R., M. Robey, and S. Nair, *Integrating Design Formalisms in Software Engineering Education*, in *Proceedings of the Seventeenth Conference on Software Engineering Education and Training*. 2004, IEEE: Norfolk, VA, USA. p. 78-83.

136. Wahl, N.J., *Student-Run Usability Testing*, in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*. 2000, IEEE: Austin, TX, USA. p. 123-131.
137. Wilde, N., et al., *Some Experiences With Evolution and Process-Focused Projects*, in *Proceedings of the Sixteenth Conference on Software Engineering Education and Training*. 2003, IEEE: Madrid, Spain. p. 242-250.
138. Williams, L., *Integrating Pair Programming into a Software Development Process*, in *Proceedings of the Fourteenth Conference on Software Engineering Education and Training*, D. Ramsey, P. Bourque, and R. Dupuis, Editors. 2001, IEEE Computer Society: Charlotte, NC. p. 27-36.
139. Wohlin, C., *Meeting the Challenge of Large-Scale Software Development in an Educational Environment*, in *Proceedings of the Tenth Conference on Software Engineering Education and Training*. 1997, IEEE: Virginia Beach, VA, USA. p. 40-52.
140. Wohlin, C. and B. Regnell, *Achieving Industrial Relevance in Software Engineering Education*, in *Proceedings of the Twelfth Conference on Software Engineering Education and Training*, H. Saiedian, Editor. 1999, IEEE Computer Society. p. 16-25.
141. Yourdon, E., *Preparing Software Engineers for the 'Real World'*, in *Proceedings of the Fifteenth Conference on Software Engineering Education and Training*. 2002, IEEE: Covington, KY, USA. p. 2-2.