Transferring Software Development Knowledge and Skills in the Academia: A Literature Review

Maíra Marques, Sergio F. Ochoa

Department of Computer Science
Universidad de Chile
Santiago, Chile
{mmarques,sochoa}@dcc.uchile.cl

***Abstract* – There is a well-known gap between what is taught in the academia on software development and what the industry needs. Trying to deal with this gap the universities have been using several mechanisms to transfer software development knowledge and skills to future engineers. Although there is no clear recipe to do that, it seems to be a consensus that such transference should be done through practical activities, e.g. developing software projects during software engineering courses. These activities help students to assimilate, link and apply software engineering concepts, and contributes to reduce the mentioned gap. This article presents a survey of the strategies used by the universities to try and deal with this challenge.**

Keywords: software engineering education, software development, teaching approaches.

I. INTRODUCTION

Software engineering (SE) is a discipline oriented to apply a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software [1]. As any engineering discipline, it is tightly linked to the industry; therefore, the software engineering curricula used in computer science programs need to prepare students to the world of industrial software development [2].

Fox and Patterson [3] pointed out that while new college graduates are good at coding and debugging, employers complain about other missing skills that they think are equally important (e.g. working with non-technical customers, working in teams, etc.). The usual faculty reaction to such criticism is that they are trying to teach principles.

This is well known that the industry needs people able to work in teams to develop and maintain software. However delivering young professionals that are proficient in technical and operative issues is a great challenge for educational institutions [4] [5] [6] [7]. Begel and Simon [8] pointed out that universities are striving to prepare students for the industrial scenario, and allowing them to become lifelong learners able to keep pace with the SE advances. Typically the universities tend to adequately teach the technical aspect of SE, but do not address operative issues such as managing projects/risks, defining/adjusting software processes, and performing team work.

Some universities worried about this gap have changed their computer science curricula to include practical activities and real clients in software engineering courses [9] [10] [11]. Others institutions use simulated projects trying to reproduce real life environments of software development [12] [13] [14]. However none of these approaches have a clear and successful solution to address the stated gap.

This article presents a literature review about the approaches used by the institutions to transfer software engineering knowledge and skills to their students. The article also discusses the different approaches used to try to identify common strategies, and their potential strengths and weaknesses.

The structure of this paper is as follows. Section II presents the state of software engineering education, its challenges and causalities. Section III introduces the general strategies used to deliver software development knowledge and skills to the students. Section IV classifies and characterizes the several strategies reported in the literature. Section V briefly describes the software processes used to guide software development activities in the academia. Section VI presents a discussion based on the analysis of the results. Section VII presents the conclusions of this study.

II. CHALLENGES TO TEACH SOFTWARE ENGINEERING

The wide use of technology creates very complex situations that professionals in IT need to be able to manage (Callahan 2003). Typically, most of the situations that engineers find in practice differ from those they have found previously. Therefore, engineers must be able to use engineering concepts that they have learned during their education to create solutions for real-world problems [4]. However, it is well known that the software industry is usually unsatisfied with the level of real-world readiness possessed by new engineering graduates entering to the workforce [4] [16] [17].

Parnas [18] defined three steps required to produce software

engineering programs: (1) to define the possible tasks that a software engineer is expected to perform, (2) to define a body of knowledge required for the software engineer, then (3) to transfer such knowledge as a training program. This definition clearly identifies the need of transferring the most important knowledge to the students and also the skills required to use it in future developments. It is aligned with the role that Denning states for software engineers [4]. These definitions suggest that we need to address two main challenges: *(1) to identify and transfer the key knowledge to the students*, and also *(2) to encourage students to develop the skills required to apply such a knowledge in real-world work scenarios*. According to Parnas [18] [16] [17], these challenges should be addressed through hands-on experiences that expose students to some likely problems before they become stranded on their own.

Concerning the first challenge the IEEE develops periodically new versions of the Software Engineering Body of Knowledge (SWEBOK) [19], which helps instructors and institutions to identify the key knowledge. However, the most challenging part is not to identify the relevant knowledge, but to transfer it appropriately. Lethbridge et al. [20] identified some points that would help address this knowledge transference; e.g. performing practical activities linked to the industry reality, and ensuring that SE educators have the necessary background to guide the process.

There are also other researchers that identified challenges on other aspects of the transference process: teaching software process with traditional methods (expositive lectures) [21] [22], educational challenges [23] and features of the development process or evaluating the work performed by the students [2] [24]. Next, these issues are briefly discussed.

- *Software development is non-linear* – Although activities, tasks and phases in the development process can be repeated, they never occur in the same way and obtain the same results [21].
- *Software development is not a straight science* – The process involves several intermediate steps and continuous choices among multiple viable alternatives. Difficult decisions must be made on-time, tradeoffs must be considered, and unanticipated events and conflicts must be handled [21]. These features generate several risks to the work of both, instructors and students.
- *Software development is sensitive* – This process may exhibit dramatic effects with non-obvious causes [22]. This feature makes that the risk management must be done during that process and also that it should involve experienced people, which represents a big issue in the SE educational scenario.
- *Software development is not hierarchical* – The development process involves multiple stakeholders, therefore decisions are made by many people that are not necessarily part of the development team [21]. Managing these situations properly requires an important communication and coordination link among members of the development team and also with the stakeholders.
- *Software development does not make everybody involved happy* – The projects often have multiple, conflicting goals, tradeoffs between aspects such as quality versus cost, timeliness versus thoroughness, or reliability versus performance [22]. These regular conflicting situations require the cooperation of the involved people to overcome them quickly. In other words, they become a waste for the project.
- *Software development requires teamwork* - Software projects are rarely based on individual effort; therefore people must work together in teams to achieve a specific goal. Unfortunately, interaction and cooperation among people are not usual strengths of computer science students [23].
- *Software development requires project management capability* - A well-managed project successfully delivers its results within the given constraints and resources. Properly managing the social and engineering aspects of a project requires capabilities and experience that are not usually present in computer science students [23].
- *Software development requires a fear evaluation of team members* - Evaluating students participating in teams is a complex task [2]. If students are graded equally according to the project results, we would be assuming that everyone worked and contributed equally, which is usually false. If students are graded separately, it is not evident how to grade the team members contribution, due the instructor having little visibility of what really happens inside a team.
- *Software development is not about writing code*. Frequently, the students believe that developing software is tightly related to writing the code that will be part of the product. Carrington and Kim [25] stated that one of the major challenges teaching software engineering is helping students understand the difference between the small programs that they write during graduate school and the software products that they have to address in the industry.
- *Software development requires a professional attitude of the students*. Chen and Chong [24] identify three harmful syndromes, which are tightly connected to the "student condition" of the developers in the academia. The first one is the well-known "free-rider syndrome" [26]. The second is "Wyatt Earp syndrome" [27], that becomes present when a student makes a project commitment with the instructor without having first discussed it with other team members. The third one is the "student syndrome" [26], which indicates that many people do not fully apply themselves to a task until the last possible moment before a deadline.

The issues presented in this section do not represent a full list, but are presented to illustrate some of the reasons making SE education particularly challenging for both students and

instructors.

Pollice [28] brings to the discussion some aspects that have been barely discussed in SE education: What are we really doing teaching or training? Clearly, teaching is closest to the identification and delivery of key SE concepts to the students, and training is related to the use of such concept in practice. Transferring the SE knowledge (i.e. teaching) and the skill required to use it in practice (i.e. training) seem to be mandatory in SE education. In fact, one of the most prestigious conferences in this area, the Conference on Software Engineering Education and Training (CSEET), clearly identifies these two aspects in the research topics that it addresses.

Pollice [29] also opens the discussion as to what is the subject that we are really delivering, software development or software engineering? In this article we call "software development" a systematic way of developing software using a process that considers the typical software life cycle. Just to relate these concepts, we define software engineering as the application of engineering principles to the development process and also to the product being developed.

The next section introduces the strategies reported in the literature to teach and train students in software development and software engineering.

III. STRATEGIES TO TEACH SOFTWARE ENGINEERING/DEVELOPMENT

There are a lot of different initiatives that try to address the problem of teaching SE. In this session we will discuss these initiatives, their usage and results. The major teaching categories found were the following: *traditional (expositive) methods*, *problem based learning methods*, *hands-on methods* (hands-on labs/simulated labs) and *case studies*. These strategies are explained in the next sections.

A. Expositive Methods

The expositive (or traditional) approaches used in SE education, involve a lecturer standing in front of students, with the course content divided into a number of topical lectures [22] [12]. The instructor generally tries to stimulate students to discuss the material delivered in the lectures. Sometimes the lecturer assigns relevant readings – normally textbooks or research papers to enrich these discussions and the material comprehension.

The evaluation is based on students understanding of the topic and/or their capability to apply the concepts in a mock software project [12]. This passive teaching strategy seems to be very distant from those required by future engineers to develop software in the industry.

B. Problem-Based Learning Methods

The *Problem-Based Learning* (PBL) method is characterized as a learning approach in which students are given more control on their own learning than in traditional approaches. In PBL the students are asked to work in small groups and acquire new knowledge only as a necessary step in solving problems that are representative of the professional practice [12]. The main idea is that these real situations are used to present challenging operational problems which generate learning outcomes that incorporate the professional knowledge, skills and competencies required by graduates.

Richardson et al. [12] report a two-year research project with graduate students, where at the beginning of the lectures, a real-world problem is presented to the classroom. The students had to perform roles as members of a development team and achieve a solution to the stated problem. These researchers also reported the use of PBL as a positive experience for both instructors and students. It provided students with the experience of solving a real-world problem and the opportunity to understand the relevance of the software quality and its potential effects.

C. Hands-on Methods

Ma and Nickerson [30] pointed out that "hands-on experience is at the heart of science learning". Hands-on activities involve a physically real investigation process. They also advocated that hands-on labs provide the students with real data and unexpected clashes between the disparity of theory and practical experiments, which are essential for students learning. It can be a challenging and realistic opportunity for students to apply the concepts of software engineering practice and process that they learn as part of their studies [9]. It is also possible to divide this hands-on experience among: *labs*, *simulated labs* (imitations of real experiments, where the infrastructure required is not real, but simulated by computers) and *remote labs* (characterized by mediated reality).

Most experiences on hand-on methods are focused on training the students in software development. However, there are also some experiences focused on engineering software products; particularly those using a pre-established software process.

Hands-on Labs

Sebern [9] reports the experience in the Real Lab, which incorporates real clients in the teaching of practical software engineering. This researcher recommends the use extensions of legacy systems as the goal of these labs, rather than engineering new solutions from scratch. He contends that the development of projects from scratch constrains the size of the software system being developed and increases the risk that the final products are not functional or incomplete.

The Team Software Process (TSP) is the basis of the software process used by students in these labs. He reported that the students were capable of linking together theory and practice, and students and clients participating in the Lab

obtained useful benefits.

Andrews and Lutfyya [11] used the idea of software maintenance to teach software engineering. They defined software maintenance as the modifications that are made to software after its initial release. They based their decision on the fact that many organizations are reluctant to completely replace older software.

Tvedt et al. [31] created a kind of software factory, based on the idea of hands-on experience. Here, the course involves a two-semester project (developments from scratch and also maintenance projects), real clients and the students playing a role as team members. They reported that the beginning was hard for instructors and students, but after getting things started, the course ran smoothly and initial reports from students seem to show that the course was working as planned.

Cagiltay [32] chose the development of games as an appropriate scenario to teach SE. Games or game-inspired exercises have been used in programming courses for sometime because they have the potential to motivate learners. Typically, students are end-users of games, therefore they usually know which is the business core of the product under construction. As users, students are able to compare different user interfaces among games that they have experienced over the years [33]. This situation improved the self-motivation during the experience, and allowed that some students to continue working on their project after finishing the course.

Jaccheri and Osterlie [34] used the participation of students in open source software projects as a scenario to teach software engineering. In those experiences the researchers found that the documentation of some projects is not strong enough to help young students participate, and that the learning curve of students can be a problem if the course is too short. However, they also reported that the students liked the experience and some of them continued supporting the open source projects.

Similar to the previous experience, Allen et al. [35] also used open source software projects as a teaching/learning scenario. However in this case the instructors chose the projects and documented them before the students began their development. They reported that the student productivity was below their expectations and that the students' learning curve was steeper than intended.

Germain et al. [36] performed a hands-on software course where all teams had to develop the same project using the *UPEDU process* (Unified Process for EDUcation). That process is a customized version of the Rational Unified Process (RUP) for education [37]. After several experiences these researchers concluded that some disciplines dominated the team's effort and that the learning curve of the process only let the development begin after the middle of the course. Therefore, the students obtained very good interim artifacts (e.g. requirement list and components design), but they were not able to fully develop everything they were supposed to.

Moore and Potts [38] created *The Real World Lab* that was designed to emulate an industrial organization as closely as possible. Projects are accepted from industry sponsors, who in turn act as "customers", providing consulting, reviewing, direction and resources to the students. The authors reported that at initial stages of this experience they began using the "Mini-Task" model based on a waterfall lifecycle, but after the first iteration, the students decided to adopt an informal ("ad hoc") process where they were able to include all the processes and formalism they thought would be an improvement. In a later work, Moore and Brennan [39] used the CMM model to evaluate the "ad hoc" process created by the students and they found out that such a process addressed the key process areas required for a CMM level 3 certification. In both papers the authors to indicate the importance of hands-on experiences and highlight how these experiences help students to be much more confident and motivated to finish their undergraduate studies.

Hands-on Simulated Labs

Vaughn [40] reported an experience in which he used a simulated process as the main instrument to teach software engineering. The experience involved real clients for the student projects, and the students had to act as a real company trying to develop real software. However the students did not have to implement the product, but only carry out all the process activities considered in a real software company (e.g. requirements specification, design, test plan, documentation and project/budget management). The author reported that the main challenge was to find clients willing to participate, even knowing that they will not have any functional software at the project end. Though these experiences were time demanding and not pleasant for students, but the students valued them.

Drappa and Ludewig [41] also developed a simulation game named *SESAM (Software Engineering Simulation by Animated Models)* to teach Software Engineering principles and practices. Each student must play the role of the project manager, and control the project simulation through a textual interface. In order to manage the simulated project, the student is able to hire team members, assign tasks to them and monitor the project progress. The player's aim in this simulation game is to complete the software project successfully. When the project is completed, the student is able to analyze his/her performance using an analysis tool. This enables students to understand the overall project results, as a consequence of their actions and management decisions. The main limitation of these types of simulations games is that they have to be fun and also provide an interesting educational value. It is also important to find a balance between the level of challenge offered and the level of skill required. An unbalanced contest with low challenge and high skill games result in boredom; and high challenge, with low skill games generates anxiety [21].

Navarro et al. [22] used a game-based simulation tool called *SimSE*, a computer based environment that allows the creation

and simulation of software development processes. It is a single player game in which the students play the role of a project manager that is in charge of a development team. These researchers reported one semester of usage of this game and indicate that sometimes the students do not really know why they were not able to finish the game properly, but after the instructor's explanation, they understand what went wrong.

Shaw and Dermoudy [42] developed a simulation game for two software development lifecycles. The game allows students to gain experience managing a simulated software development project in an environment that is both, graphical and entertaining. These researchers point out that using simulations allow the involved people to experiment with the project, which would not be possible in a real-world scenario. The likely cost of implementing changes, potential consequences or even danger that could result from experimenting with a real system make simulations an attractive alternative [43]. The Shaw and Dermoudy article concludes that students benefit from the experience of playing the simulation game. Students were successfully able to identify the reasons why their simulated projects had failed or succeeded, and in the case of failure determine a strategy that would allow them to avoid this in the future.

Hands-on Remote Labs

Bayliss and Strout [44] report an experience of a hands-on remote lab using mediated reality (i.e. all meetings required to perform the project were computer-mediated). The course was delivered online and roles were assigned to students. Each role was played in pairs. The authors tracked the students after the course and found that the students, who performed well on the course, were able to advance (on placement tests) quickly in their undergraduate courses.

D. Case Studies Method

Bernstein and Klappholz [45] created the idea of live-thru case stories, which is a teaching method developed specifically to enhance students' visibility about the software development process. The method consists of discussing failed case stories and to recognize the oversights and mistakes of others. The authors found that discussing failed case stories and having the students read similar case studies only amounted to them recognizing the oversights and mistakes of others; but did not help them recognize their own mistakes. This method is focused on training students to perform software development, and it seems to be a very powerful experiential learning tool. The authors report that the analysis of real situation failures allow students to internalize several issues, such as the need for a software process, the team member attitudes towards customer interaction, the developer's responsibility towards customers. However, live-thru case stories have several limitations, for example they are time consuming.

IV. COMPARING TEACHING/TRAINING STRATEGIES

This section presents the results of a literature review on strategies to deliver software engineering knowledge in the academia. This review classifies the proposals according to the approaches described in the previous section. It also presents the relevant features that describe these proposals, such as the class size where the course was taught, size of the development team, course duration, how the course was graded, the target population to whom the course was focused on, the type of development process used by the teams, and the presence of interim checkpoints/real clients/coaches during the project.

This characterization of the proposals helps people illustrate what approaches are available and which are the excepted outcomes for them. Based on that, a decision about which strategy to adopt for enhancing practice in SE courses can be made in a much more straightforward manner. The tables presented in this section include the following information:

- *Reference* – bibliographic reference;
- *Approach Type* (just Table 3) – types of approach used to teach software engineering/development (discussed in the previous section);
- *Software Process* – type of software process used to guide the project;
- *Class Size* – number of students enrolled in the course;
- *Team Size* – number of students in each team;
- *Experience Duration* – time frame of the course;
- *Group Formation* – strategy used to make students part of a team;
- *Roles* – presence of roles in the development team;
- *Grades* – strategy used to grade the students;
- *Target Population* – target population to whom the course was taught (e.g. undergraduate, graduate or mix of students);
- *Development Team Type* – the type of development team used in the course (e.g. co-located or distributed teams);
- *Checkpoint* (Iteration) – presence of checkpoints in the course (type and amount);
- *Coach* – presence of a coach that supports the teams during the development process;
- *Real Client* – presence of a real client that interacts with the development team;
- *Documentation* – Type of documentation required for the project;
- *Observation Time Span* – time of observation reported by the researchers.

Unfortunately not all the experiences reported in the literature indicate (or allow us to infer) the value for the features presented above. In order to clarify the semantic of the symbols used in the tables, next we indicate the meaning of each of them:

- The symbol "-" indicates that information was not available to determine or infer the value of a feature;
- The symbol "x" means that an evaluated feature does not apply for a particular proposal;
- The symbol "()" indicates that the feature value was inferred based on the information given by the researchers reporting the use of a particular teaching approach.

The approaches specified next are more focused on teaching software development than software engineering.

1) Traditional Approach

In Table 1 we mention a survey done by Ford [46] reporting on the teaching of software engineering in 1994, where at least 14 American Universities were teaching software development in the traditional way. Yanchum [47] reported that the traditional approach was still in use in 2011.

Table 1 - Traditional Approaches used for Teaching Software Development

Reference	Software Process	Class Size	Team Size	Experience Duration	Group Formation	Roles	Grades	Target Population	Development Team Type	Checkpoint (Iteration)	Coach	Real Client	Documentation	Observation Time Span
Ford [46]	x	-	-	12 weeks	x	x	Exams and homework	Several	x	x	-	x	x	-
Yanchum [47]	x	-	-	-	-	-	-	Undergraduate	x	x	-	x	x	-

The traditional approach is the one being used to conduct instructional processes in almost every knowledge area. The approach considers delivering the knowledge through expositive lectures, and tries to apply such knowledge to the practice through toy exercises. Clearly this approach does not allow students to address the software development challenges that they are going to see in the industry. This could be the reason why there are only a couple of papers reporting the use of traditional approaches to teach software engineering.

2) PBL (Problem Based Learning) Approach

Table 2 shows two examples of the PBL approach in which small teams (from 4 to 6 students) were involved in projects as part of graduate and undergraduate courses. The use of this approach to teach SE is quite new, since all the papers reporting these experiences are recent. Typically, in a PBL approach the students are divided in teams, and each team member has the responsibility of particular tasks. In this approach there are no roles, only tasks. In each PBL the task to be performed changed and the team member responsible for it also changed.

Table 2 - PBL Approaches used for Teaching Software Development

Reference	Software Process	Class Size	Team Size	Experience Duration	Group Formation	Roles	Grades	Target Population	Development Team Type	Checkpoint (Iteration)	Coach	Real Client	Documentation	Observation Time Span
Gold [51]	-	100	6	12 weeks	Instructor allocations	No	Peer assessment	Undergraduate (2^{nd} year)	-	Report	No	No	Minutes, reports (tender, background, design, progress, reflection)	1 year
Richardson, et al. [12]	-	14	4	12 weeks	Instructor allocation	No	Team project, exam	Graduate (Master)	Distributed	None	No	Instructor as client	Minutes, reports	2 years

3) Hands-on Methods

This section reports several hands-on methods to teach software developments. These methods are those introduced in the section III.C.

a) *Hands-on Labs*

The hand on labs is an approach in which the main focus is training, although in some cases there is a teaching process to deliver SE knowledge. Table 3 reports the use of the hands-on lab approach for teaching software development. The literature reports the use of this approach mainly between 1994 and 2009. The average size for the development teams used in hands-on labs was 4-6 students; 40% of the experiences covered a duration of one semester (12-18 weeks), and 47% of them involved experiences of one academic year (26 weeks approximately).

The teams can be formed by student choice (33%), by instructor allocation (33%), in the final 33% the information was not available. Students are normally graded by peer assessment (40%), and courses are normally utilized to teach undergraduate students (67%), but 33% are for graduate students. The development teams normally perform co-located work (40%) more than distributed work (20%). Most projects involve a real client (67%), and a few of them have no client or had an instructor acting as client (17% each).

The observation time span reported was of one year. It is worth mentioning that most teams have a maximum of 6 members. This happens because the bigger the team, the bigger their vulnerability and cost required to perform the communication and coordination activities. The experiences over 16 weeks (or more than one semester) use roles. Researchers reporting those experiences indicate that at some point in the project the students switch roles. Some researchers do not report the use of roles, but they mention that each student is in charge of specific tasks, which is almost equivalent to using roles in the development team.

Table 3 - Hands-on Approaches for Teaching Software Development

Reference	Project Type	Software Process	Class Size	Team Size	Experience Duration	Group Formation	Roles	Grades	Target Population	Development Team Type	Checkpoint (Iteration)	Coach	Real Client	Documentation	Observation Time Span
Robillard, Kruchten and D'Astous [37]	-	UPEDU	30	6	13 weeks	-	Yes	-	Undergraduate (3rd. and 4th. year)	-	According to UPEDU process	-	No	UPEDU artifacts	1 year
Groth and Robertson [10]	-	CMM based	-	4	Three semesters	-	Yes	Peer assessment	-	-	Each documentation delivered is a checkpoint	Senior students coaches new ones and instructors supervises	Yes	Proposal feasibility, requirements, prototype, design, user documentation	2 years
Rico and Sayani [14]	-	Agile Mix	15	5	13 weeks	Students choice	No	Peer assessment	Graduate	Co-located	3 Iterations	Assistant professor	Instructor as client	-	-
Port and Boehm [49]	-	MBase	-	-	Two semesters	Instructor allocation	Yes	-	Graduate	Co-located	3	-	Yes (Faculty Departments)	Use cases, GUI	2 years
Moore and Potts [38]	-	-	37	-	Three quarters	-	Yes	-	Undergraduate (3rd. and 4th. year) and Graduate	Co-located	End of each quarter	Graduate students acts as coaches	Yes	Oral and written reports	-
Halling, et al. [57]	-	UP	250	6	One year	Instructor allocation	Yes	-	Undergraduate (2nd. year)	-	4 Phases	-	Yes	Vision, requirements, architectural design, user documentation, final presentation	2 years
Sebern [9]	-	TSP – "ad-hoc" - CMMI	-	4-6	Three quarters	Instructor allocation	Yes	Peer assessment	Undergraduate (3rd. and 4th. year)	Co-located	According to TSP process	Instructor	Yes	According to TSP process	2 years
Villena [54]	-	Agile	20	4-6	16 weeks	Time allocation	No	Peer assessment, instructor assessment, client assessment	Undergraduate (6th. year)	Co-located	3 Formal technical reviews	Instructor and experts	Yes	Presentations	6 years
Wohlin [48]	Software Maintenance	PSP adaptation	100-125	14-19	7 weeks	Students choice	Yes	-	Graduate (Master)	-	4 Phases	External professors	Clients idea, but the instructor acts as the client	Requirements analysis, design specification, planned effort,	-

Gehrke, et al. [13]	Software Maintenance		200	5	One semester (18 weeks)	Students choice	Only 2 roles	-	Undergraduate (2nd. year)	Distributed	Every week	-	Instructor as client	Final report	4 years
Tvedt, Tesoriero and Gary [31]	Software Factory	Not Agile	25	-	Eight semesters	Vary	Yes	Peer assessment + manager grades	Undergraduate	Co-located	End of each semester	Instructor	Yes	-	-
Cagiltay [32]	Computer Game Development	Not Agile	125	-	26 weeks	Instructor allocation	No	Project grades, final exam	Undergraduate (4th. year)	-	-	None	No	Reports, requirements, UML diagrams	3 years
Ellis, Morelli and Hislop [53]	Open Source Software	(Agile)	9	-	15 weeks	Students choice	No	Peer assessment	Undergraduate	Distributed	-	-	Yes (open source community)	Software requirement specification, software design specification	2 years
Andrews and Lutfyya [11]	Open Source Software	(Agile)	-	6	26 weeks	Students choice	No	-	Graduate (3rd. year)	Distributed	Middle and final presentation	Instructor (twice during the whole project)	Yes	Progress reports and presentations	-
Carrington and Kim [25]	Open Source Software	(Agile)	260	3-4	One semester (13 weeks)	-	-	-	Undergraduate (2nd. year)	-	4 assignments	-	Yes (open source community)	Reports	-

b) Hands-on Simulated Labs

The hands on simulated labs approach is mainly concerned with training software development and not focused on transferring SE knowledge. Depending on the simulation goal, the instructor asks the students to perform particular tasks or reach a certain goal. Table 4 shows two examples of the hands-on simulated labs approach involving several team sizes. This approach has been used for some time only in graduate courses and there are no new reports of its use in the academia.

Table 4 - Hands-on Simulated Labs Approaches for Teaching Software Development

Reference	Software Process	Class Size	Team Size	Experience Duration	Group Formation	Roles	Grades	Target Population	Development Team Type	Checkpoint (Iteration)	Coach	Real Client	Documentation	Observation Time Span
Drappa and Ludewig [41]	-	-	9	One semester	Individual	No	-	Undergraduate (3rd. year)	Co-located	-	Instructor	No	-	1 year
Vaughn [40]	(Agile)	20	4-6	One semester	Instructor allocation	-	Peer assessment	Undergraduate	Co-located	5 Phases	-	Yes	Reports	1 year

c) Hands-on Remote Labs

The hands-on remote labs are focused on teaching software development, the knowledge transfer is only concerned with the use of technology. During these labs the students worked in teams, where they all have pre-assigned tasks to be performed. Table 5 shows one example of the Hands-on Remote Labs Approach. It was a fast course (summer session) which lured students to enroll in computer science majors. It was considered a very good experience that fulfilled the chosen goal.

Table 5 - Hands-on Remote Labs Approaches for Teaching Software Development

Reference	Software Process	Class Size	Team Size	Experience Duration	Group Formation	Roles	Grades	Target Population	Development Team Type	Checkpoint (Iteration)	Coach	Real Client	Documentation	Observation Time Span
Bayliss and Strout [44]	(Agile)	-	-	Summer Course (10 weeks)	Instructor allocation	No	Project grades	Undergraduate	Distributed	-	-	No	-	1 year

4) Case Studies Approach

The case study approach is focused on delivering software engineering knowledge. During these experiences the students have specific tasks to perform in an assigned case study. Student tasks or roles can change when they address a new case study. Table 6 indicates the experiences reporting the use of the case studies approach. This approach usually involves small teams, and it can be applied to both co-located and distributed development teams.

Table 6 - Case Study Approaches for Teaching Software Development

Reference	Software Process	Class Size	Team Size	Experience Duration	Group Formation	Roles	Grades	Target Population	Development Team Type	Checkpoint (Iteration)	Coach	Real Client	Documentation	Observation Time Span
Bernstein and Klappholz [45]	x	-	2	-	-	-	-	Undergraduate	Co-located	x	x	x	-	1 years
Budimac, et al. [52]	x	80-100	3-5	One semester	Students choice	Yes	Peer assessment	Undergraduate ($4^{th.}$ year)	Distributed	x	x	x	Reports	2 years

5) Summary

Observing the results of this literature review we can say that there are some approaches that were poorly utilized or reported, for example the hands-on remote labs (one report) and the simulated labs (two reports in the same year). We hypothesize these results are mainly because of the complexity and effort required to implement those labs. The case study approach was not a success either, but it has two different reports with a ten-year difference between them, meaning that it may still be relevant. The PBL approach can be considered a new proposal that considers two recent reports (2010 and 2011). The hands-on labs approach was the most reported (15 papers), and it was also the proposal that has been used the longest (the reports span from 1994 to 2008).

Analyzing the challenges presented in Section II, some authors used a defined software process [9] [48] [37] [10] [14] [49] [50] to deal with most of those challenges. Other researchers preferred to deal with that in ad-hoc ways [13] [32] [40] and some of them did not mention any concern or did not explicitly mention anything about it. This was the case for at least five different challenges: software development is non-linear, software development is not a straight science, software development is sensitive, software engineering is not hierarchical, software engineering does not make everybody involved happy.

The people problem (i.e. teamwork) was a concern for almost all the researchers. Some of them tried to deal with this issue by choosing the team composition [12] [51], and others tried to address it allowing the students to choose their own teammates [52] [53] [13]. Moreover, some of them showed no concern with regard to this matter [11] [41].

Several articles highlight the importance of counting on a coach that guides the students and helps them to work as a team [38] [10] [9] [62].

Teaching students how to manage a project was defined as a goal by several researchers [9] [10] [11] [31] [32] [49] [38]. In order to do that, they created different kinds of hierarchies inside the teams, where students have to learn the basic responsibilities of traditional roles (e.g. developer, tester, analyst, designer, and project manager). Thus, the students will understand the importance of each role, the best way to balance them and coordinate their activities. There are some reported cases where, because of the chosen methodology, it was impossible to teach project management [52] [53] [11]. Drappa and Ludewig [41] reported the use of a specific project manager simulator to make students aware of the relevance of the subject. The other researchers tried to teach something about project management (since it is embedded in software development), but there is no formal teaching of the subject.

The majority of the authors addressed the last challenge (i.e. students evaluation) by using peer assessment as a way to evaluate the student's work. They acknowledged the difficulty of evaluating students working in teams, and identifying the syndromes mentioned in Section II. However, some researchers still prefer the traditional methods to evaluate students and do not worry about the topic [12] [32] [44].

V. Software Development Processes

As discussed in the introduction, the software processes can help improve the efficiency of development teams. Software development processes are "used as guidelines or frameworks to organize and structure how software development activities should be performed and in what order" [54]. These guidelines have the aim of transmitting software engineering knowledge and skills to the students. Therefore, it is expected that these processes carry some relevance in SE courses.

Various researchers recommend particular approaches to deliver theoretical knowledge on software engineering, but few of them indicate how to use such knowledge in real-world projects. According to this literature review, the processes being used in SE courses are the following: UPEDU, CMM, PSP, TSP and Agile. Next we briefly explain each of them.

- *UPEDU* – Unified Process for EDUcation, an academic customization of the RUP 2000. UPEDU is based on the concept that a software development is a collaboration process between abstract active entities called roles. This process uses 37% of the RUP artifacts and 10% of its guidelines.
- *PSP* – Personal Software Process is a structured process that is intended to help developers understand and improve their performance by using a disciplined and data-driven procedure. The PSP was created by Humphrey [55] to apply the underlying principles of the CMM (Capability Maturity Model) to the software development practices of a single developer. One of the core aspects of the PSP is the use of historical data to analyze and improve the developers' performance. PSP data collection is supported by four key elements: scripts, measures, standards and forms.
- *TSP* – Team Software Process, provides a defined operational process framework that is designed to help managers and engineers organize projects and produce software products. The use of TSP requires that the users have already learned PSP. The TSP software development cycle begins with a planning process called the *launch*, which is led by a coach who has been specially trained to do so. The launch is designed to begin the team building process, and during this period, teams and managers establish goals, define roles, assess risks, estimate efforts, allocate tasks, and produce a team plan. During the next stage (*execution* phase), developers track the planned and actual effort/schedule/defects, meeting regularly to report the project status and revise plans. A development cycle ends with a *post mortem* to assess the team performance, revise planning parameters and capture lessons learned for

process improvement [56].

- *MBASE* – Model Based Architecting and Software Engineering. The MBASE approach is a tool that helps manage the client expectations. Port and Boehm [49] have identified what they call *simplifiers* and *complicators:* things that make life easier or harder, for both developers and customers. Simplifiers and complicators typically affect the communication between customers and developers. These key elements are organized and classified using domain-specific headings and they are used to help customers understand the problems developers try to face, and vice versa.

- *UP* - Unified Process is considered a generic process framework that uses UML. Halling et al. [57] calls UP as a state-of-the-art and a state-of-the-practice software process framework, mainly because it is a use-case driven, architecture centric, iterative and incremental process.

- *Agile* - Agile methods are contemporary software engineering approaches based on teamwork, customer collaboration, iterative development, people, process and technology adaptable to change [58]. They are a halt of management thought, predating the industrial revolution and use craft industry principles (like artisans creating made-to-order items) for individual customers [59].

Umphress et al. [60] reported the history of teaching software development process in Auburn University. In 17 years of teaching software engineering, they tried five different approaches and compared the advantages and disadvantages of each one. First they tried an ad-hoc process for ten years, and found out that the product seldom met client expectations. To solve the problem they adopted the MIL-STD-498 process for two years, eventually deciding to quit because the process was centered only in documentation. Then, they tried the IEEE 1074 for four years, but they had to drop it because of the complexity of the process. After they tried the TSP for only a year and realized that, again, the paperwork was overwhelming to the students. Finally, they tried the Extreme Programming approach for one year, and the informality was the biggest disadvantage that they found. They concluded that teaching students a process is very important, but they emphasize that one must weigh the process against students' abilities, expectations and tolerance level. They point out that none of the software processes they have tried are bad; they just needed a lighter process to meet their students' culture and environment.

VI. Discussion

The aim of this survey is to report on the reality of what was found about development processes and on the teaching/training strategies to address software engineering education. In Section I we mentioned some difficulties that challenge the teaching of software engineering. In Section IV, it was discussed how other works have addressed the already mentioned challenges.

The hands-on labs approach was the first one reported in the literature of software engineering education which addressed the software development process in the Academia. Besides the fact that it was the most used and the most extended in time use, the first report was in 1994 and the last one was in 2009. The other approaches were much less reported and its use corresponds to short time periods. For example, the articles reporting the PBL approach appeared during 2010-2011; hands-on simulated labs during 2000; hands-on remote labs during 2006, and the only exception were case studies reported being used in 2001 and in 2011. The first approach researched on the matter was the most relevant.

It seems that there is a consensus among researchers that it is important for students to feel the reality of developing software before they finish their courses; since the majority of the work reported by researches uses the hands-on labs approach.

There is not a right or wrong approach for teaching software development in the Academia. There are just distinct flavors of approaches to deal with the stated challenge. Each instructor then has to choose which one will adhere more to his specific teaching scenario (i.e. public, course duration, university grading politics, etc.). This paper reports on the reality of what is being done so far. This is not a closed matter because there are many ways to teach any subject and at any given time someone can initiate a new one.

This work has also several limitations. First of all, not all universities have researchers that reported their experience developing software in the academic setting. Universities using the traditional approach may think that what they are doing is antiquated and it did not need to be reported at all.

Another limitation worth mentioning is the source of the articles analyzed in this work. These sources were the ACM Digital Library, Google Scholar and IEEE Digital Library. A total of 150 articles were analyzed, and many of them were included in this survey.

VII. Conclusions

There are a lot of ideas on how to transfer the skills required to develop software. The time each university wishes to spend on teaching these skills also can change a lot. However, one point is clear, academia is not worried about the process used to teach software engineering to students. They are trying to teach software engineering in a "wild" way.

The industry knows of this problem and they are always complaining about that. They say that they have to teach recently graduated engineers on how to work in a software development team. The industry can be more participative and help the academia more in teaching software engineering, for their own good.

Teaching software development is not an easy task. This survey presents a series of strategies used to teach software development. The main idea of all the strategies presented here is to teach students to maintain a standard in development to achieve client goals. There are a lot of difficulties in transferring this knowledge to students because normally students are reluctant to produce documentation and work in groups, as they prefer individual homework assignments, which they can start coding right away [61].

The use of software development processes in this knowledge transfer was not so widespread as expected. Mainly it is hard to find a match between the software process weight, and the students' abilities and expectations [60].

There is a tangible need for strategies to transfer software development knowledge in the academia. The majority of strategies that exist did not prove worthwhile. After some research and publication, they are completely dropped by the researchers and put to rest in history. Tough not an easy task, I believe that it is possible to create software development processes that can be used in the academia with positive results.

ACKNOWLEDGEMENT

The work of Maíra Marques Samary was supported by the Conicyt (Chile) scholarship for PhD students.

VIII. BIBLIOGRAPHY

[1] IEEE Standards Collection: Software Engineering. IEEE Standard 610.12-1990, IEEE 1993.

[2] Chen, J., H. Lu, L. An, and Y. Zhou. "Exploring Teaching Methods in Software Engineering Education". Proceedings of the 4th International Conference on Computer Science & Education (ICSE). IEEE Press (2009): 1733-1738.

[3] Fox, A., and D. Patterson. "Crossing the Software Education Chasm". Communications of the ACM, Vol. 55, no. 5 (2012): 44-49.

[4] Denning, P. J. "Educating a New Engineer". Communications of the ACM, Vol. 35 no. 12 (1992): 82-97.

[5] Hilburn, T. B., and D. J. Bagert. "A Software Engineering Curriculum Model". Proceedings of the 29th Annual Frontiers in Education Conference. IEEE Press. (1999): 12A4-6-11.

[6] Gorla, N., and Y. W. Lam. "Who Should Work with Whom? Building Effective Software Project Teams". Communications of the ACM, Vol. 47, no. 6 (2004): 79-82.

[7] Wellington, C. A., T. Briggs, and C. D. Girard. "Examining Team Cohesion as an Effect of Software Engineering Methodology". 2005 Proceedings of the Workshop on Human and Social Factors of Software Engineering. ACM Press. (2005): 1-5.

[8] Begel, A., and B. Simon. "Novice Software Developers, All Over Again." Proceedings of the 4 International Workshop on Computing Education Research. ACM Press (2008): 3-14.

[9] Sebern, M. "The Software Development Laboratory: Incorporating Industrial Practice in an Academic Environment". Proceedings of the 15th Software Engineering Education and Training. IEEE Press. (2002): 118-127.

[10] Groth, D., and E. Robertson. "It's all About Process: Project-Oriented Teaching of Software Engineering". Proceedings of the 14th Software Engineering Education and Training. IEEE Press. (2001): 7-17.

[11] Andrews, J. H., and H. L. Lutfyya. "Experiences with a Software Maintenance Project Course". IEEE Transactions on Education Vol. 43, no. 4 (November 2000): 383-388.

[12] Richardson, I., L. Reide, B. Pattinson, Y. Delaney, and S. Seidmand. "Educating Software Engineers of the Future: Software Quality Research through Problem-Based Learning". Proceedings of the 24th Software Engineering Education and Training. IEEE Press. (2011):. 91-100.

[13] Gehrke, M., Giese, H., Nickel, U. A., Niere, J., Tichy, M., Wadsack, J. P. and Zundorf, A. "Reporting About Industrial Strength Software Engineering Courses for Undergraduates". Proceedings of the 24th International Conference on Software Engineering. ACM Press (2002) 395-405.

[14] Rico, D. F., and H. H. Sayani. "Use of Agile Methods in Software Engineering Education". Proceedings of the Agile Conference. IEEE Press. (2009): 174-179.

[15] Callahan, D. W. "Development of an Information Engineering and Management Program". IEEE Transactions on Education, Vol. 46, no. 1 (2003): 111-114.

[16] Vaughn, R. B., and J. Carver. "The Importance of Experience with Industry in Software Engineering Education." Proceedings of the 19th Conference on Software Engineering Education and Training. IEEE Press. (2006): 19-24.

[17] Huang, L., and D. Port. "Relevance and Alignment of Real-Client Real-Project Courses via Technology Transfer." Proceedings of the 24th Software Engineering Education and Training (ICSE). IEEE Press. (2011): 189-198.

[18] Parnas, D. L. "Software Engineering Programs are not Computer Science Programs". IEEE Software, Vol. 16, no. 6. (1999): 19-30.

[19] Abran, A., Moore, J.W. "Guide to the Software Engineering Body of Knowledge". 4th Edition, IEEE Press, 2004.

[20] Lethbridge, T.C., Diaz-Herrera, J., LeBlanc Jr., R.J., Thompson, J.B. "Improving Software Practice through Education: Challenges and Future Trends". Proceedings of Future of Software (ICSE), IEEE Press. (2007): 12-28.

[21] Oh, E., and A. van der Hoek. "Adapting Game Technology to Support Individual and Organizational Learning". Proceedings of the 13th International Conference on Software Engineering and Knowledge Engineering. IEEE Press. (2001): 347-354.

[22] Navarro, E. O., A. Baker, and A. van der Hoek. "Teaching Software Engineering Using Simulation Games". Proceedings of the International Conference on Simulation in Education. IEEE Press, 2004.

[23] van der Duim, L., J. Anderson, and M. Sinnema. "Good Practices for Educational Software Engineering Projects". Proceedings of the 29th International Conference on Software Engineering. IEEE Press. (2007): 698-707.

[24] Chen, C. Y., and P. P. Chong. "Software Engineering Education: A Study on Conducting Collaborative Senior Project Development". Journal of systems and Software, Vol. 84, no. 3 (2011): 479-491.

[25] Carrington, D., and S. Kim. “Teaching Software Design with Open Source Software”. Frontiers in Education. Vol. 3, (2003): S1C 9-14.
[26] Goldratt, E. M. “Critical Chain”. MA: The North River Press, 1997.
[27] Boehm, B. W., and D. Port. “Educating Software Engineering Students to Manage Risk”. Proceedings of the IEEE 23rd International Conference on Software Engineering. IEEE Press (2001) 591-600.
[28] Pollice, G. “Teaching vs. Training”. White Paper, IBM Developer Works, Mar. 2003.
[29] Pollice, G. “Teaching Software Development vs. Software Engineering”. White Paper, IBM Developer Works, Dec. 2005.
[30] Ma, J., and J. Nickerson. “Hands-on, Simulated, and Remote Laboratories: A Comparative Literature Review”. Computing Surveys, Vol. 38, no. 3. (2006): 7.
[31] Tvedt, J. D., R. Tesoriero, and K. A. Gary. “The Software Factory: Combining Undergraduate Computer Science and Software Engineering Education”. Proceedings of the 23rd International Conference on Software Engineering. IEEE Press. (2001): 633-642.
[32] Cagiltay, N. E. “Teaching Software Engineering by Means of Computer-Game Development: Challenges and Opportunities”. British Journal of Educational Technology, Vol. 38, no. 3 (2007): 405-415.
[33] Ferrari, M., R. Taylor, and K. VanLehn. “Adapting Work Simulations for Schools”. The Journal of Educational Computing Research, Vol. 21, no. 5 (1999): 25-53.
[34] Jaccheri, L., and T. Osterlie. “Open Source Software: A Source of Possibilities for Software Engineering Education and Empirical Software Engineering”. Proceedings of the 1st Workshop on Emerging Trends in FLOSS Research and Development. IEEE Press. (2007): 5.
[35] Allen, E., R. C. Cartwright, and C. Reis. “Production Programming in the Classroom.” ACM SIGCSE Bulletin Vol. 35, no. 1 (2003): 89-93.
[36] Germain, E., P. Robillard, and M. Dulipovici. “Process Activities in a Project Based Course in Software Engineering”. Proceedings of the 32nd Frontiers in Education. IEEE Press (2001): S3G-7-12.
[37] Robillard, P., P. Kruchten, and P. D'Astous. “Yoopeedoo (UPEDU): a Process for Teaching Software Process”. Proceedings of the 14th Software Engineering Education and Training. IEEE Press. (2001): 18-26.
[38] Moore, M., and C. Potts. “Learning by Doing: Goals and Experiences of two Software Engineering Project Courses”. Lecture Notes in Software Engineering Education, Vol. 750. (1994): 151-164.
[39] Moore, M. M., and T. Brennan. “Process Improvement in the Classroom”. Lecture Notes in Software Engineering Education, Vol. 895. (1995): 121-130.
[40] Vaughn, R. B. “A Report on Industrial Transfer of Software Engineering to the Classroom Environment”. Proceedings of the 13th Software Engineering Education & Training. IEEE Press. (2000): 15-22.
[41] Drappa, A., and J. Ludewig. “Simulation in Software Engineering Training”. Proceedings of the 22nd International Conference on Software Engineering. ACM Press, (2000). 199-208.
[42] Shaw, K., and J. Dermoudy. “Engendering an Empathy for Software Engineering”. Proceedings of the Australasian Conference on Computing Education. Australian Computer Society. (2005): 135-144.
[43] Seila, A. “An Introduction to Simulation”. Proceedings of the Winter Simulation Conference. IEEE Press. (1995): 7-15.
[44] Bayliss, J. D., and S. Strout. “Games as a “Flavor” of CS1”. ACM SIGCSE Bulletin Vol. 38, no. 1 (2006): 500-504.
[45] Bernstein, L., and D. Klappholz. “Getting Software Engineering into our Guts”. The Journal of Defense Software Engineering (2001): 25.
[46] Ford, G. “The Progress Report on Undergraduate Software Engineering Education”. ACM SIGCSE Bulletin, Vol. 26, no. 4 (1994): 51.
[47] Yanchum, S. "The Challenge and Practice of Creating a Software Engineering Curriculum." Proceeding of the 24th Conference on Software Engineering and Training. IEEE Press. (2011): 497-501.
[48] Wohlin, C. “Meeting the Challenge of Large-Scale Software Development in an Educational Environment”. Proceedings of the 10th Software Engineering Education & Training. IEEE Press. (1997): 40-52.
[49] Port, D., and B. Boehm. “Using a Model Framework In Developing and Delivering a Family of Software Engineering Project Courses”. Proceedings of the 14th Software Engineering Education and Training. IEEE Press. (2001): 44-55.
[50] Germain, E., and P. N. Robillard. “Towards Software Process Patterns: An Empirical Analysis of the Behavior of Student Teams”. Information and Software Technology, Vol. 50, no. 11 (2008): 1088-1097.
[51] Gold, N. “Motivating Students in Software Engineering Group Projects: An Experience Report”. Innovation in Teaching And Learning in Information and Computer Sciences, Vol.9, no. 2 (2010): 10 - 19.
[52] Budimac, Z., Z. Putnik, M. Ivanovic, K. Bothe, and K. Schuetzler. “On the Assessment and Self-Assessment in a Students Teamwork Based Course on Software Engineering”. Computer Applications in Engineering Education Vol. 19, no. 1 (2011): 1-9.
[53] Ellis, H., R. Morelli, and G. Hislop. “Holistic Software Engineering Education Based on a Humanitarian Open Source Project”. Proceedings of the 20th Software Engineering Education & Training. IEEE Press. (2007): 327-335.
[54] Scacchi, W. Process Models in Software Engineering. 2nd Edition. New York: John Wiley and Sons, Inc, 2001.
[55] Humphrey, W. “Using a Defined and Measured Personal Software Process”. IEEE Software, Vol. 13, no. 3. (1996): 77-88.
[56] Humphrey, W. “The Team Software Process”. Wiley Online Library, 2000.
[57] Halling, M., W. Zuser, M. Kohle, and S. Biffl. “Teaching the Unified Process to Undergraduate Students”. Proceedings of the 15th Software Engineering Education and Training. IEEE Press. (2002): 148-159.
[58] Cockburn, A. A Human-Powered Methodology for Small Teams. Boston: Addison Wesley, 2004.
[59] Aranda, J. "A Theory of Shared Understanding for Software Organizations." Doctorate Thesis. University of Toronto, 2010.
[60] Umphress, D., T. Hendrix, and J. Cross. “Software Process in the Classroom: The Capstone Project Experience”. IEEE Software, Vol. 19, no. 5 (2001): 78-81.
[61] Carver, J., L. Jaccheri, S. Morasca, and F. Shull. “Issues in Using Students in Empirical Studies in Software

Engineering Education". Proceedings of the 9th Software Metrics Symposium. IEEE Press (2003): 239-249.
[62] Villena, A. M. "A Model of Student Integration with Agile Software Engineering Industry" –in Spanish. Master Thesis in Computer Science, Computer Science Department University of Chile 2008.