Building XP process metrics for project-based software engineering courses

An Ju
Adnan Hemani
Joshua Zeitsoff
Yannis Dimitriadis
Armando Fox
Joshua Hug, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-39
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-39.html

May 1, 2020

Acknowledgement

This report is done in collaboration with Adnan Hemani, Joshua Zeitsoff, and Professor Yannis Dimitriadis. We also
thank Anirudh Kothapalli, Anish Saha, and Xiao Fu for their help. Thank Professor Joshua Hug for his feedback on this
report. Thank my advisor, Professor Armando Fox, for his time and effort coaching and advising me.

Building XP process metrics for project-based software engineering courses

by An Ju

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Armando Fox
Research Advisor

5-Dec-2019

(Date)

* * * * * * *

[illegible]

Professor Joshua Hug
Second Reader

5th-Dec-2019

(Date)

BUILDING XP PROCESS METRICS FOR PROJECT-BASED SOFTWARE ENGINEERING COURSES

An Ju
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720
`an_ju@berkeley.edu`

December 19, 2019

ABSTRACT

Project-based courses are widely used in software engineering curricula. In this report, we examine two issues related to project-based software engineering courses: 1) what processes should we include in the course, and 2) in order to achieve formative assessments, how can we build process metrics. In the first part, we combine a field study on professional Agile (eXtreme Programming, XP) teams and an established team process taxonomy to "proactively" select team processes to include in a project-based software engineering course. We choose processes that are 1) considered essential by professionals, and 2) *complete* with respect to coverage of the taxonomy's main categories. A case study shows that the augmented course design improves learning outcomes. In the second part, we establish the importance of measuring *processes* in project-based software engineering courses and present metrics mined from software development tools for monitoring and observing processes to facilitate teaching. A case study confirms that teams with better conformance to software development processes achieve better outcomes. Methods presented in this report can be used to improve software engineering courses. Some future studies are discussed in this report, envisioning a systematic method for improving and automating software engineering courses.

***K*eywords** Agile, eXtreme Programming (XP), project-based learning, software engineering, case study

1 Introduction

Project-based software engineering courses aim to provide learning opportunities grounded in practice [1, 2, 3]. They emphasize processes that are used by professionals to work smoothly and efficiently in teams. In this report, we focus on the widely-used Agile software development methodology [4], and specifically, the eXtreme Programming (XP) method [5]. We address two fundamental issues about project-based XP courses:

- What XP processes should we choose?
- How can we build XP process metrics?

Project-based software engineering courses are demanding for both students and instructors [6, 2, 7]. As undergraduates, students have been focusing on delivering new features, while software engineering courses transition students into professional developers who value maintainability and processes, among other attributes [8]. Studies show that professional software developers identify modern software development processes (such as testing and project management), software development tools (such as version control), and team skills (such as communication skills) as the most significant knowledge gaps that CS graduates have [9, 10]. And software engineering courses are meant to close the gaps [3]. A successful transition from students to professionals requires not only new knowledge or skills but also a change of mentality [11]. For example, new graduates could feel anxiety when they need to "waste" time learning a new codebase, while professionals recognize that time spent learning is common and worthwhile [8, 11]. This change of mentality requires carefully designed courses and close attention from instructors.

There are several methods proposed for teaching Agile or XP software development [12]. In this report, we focus on project-based courses. Project-based courses are widely used [12] and recognized as an effective method for teaching software engineering courses [13]. They provide opportunities for students to build software as professionals [13, 14]. Accordingly, project-based courses should let students use industrial processes and use processes as professionals. However, this goal is not readily achievable because of various course constraints. For example, pair programming [5] is an effective process for professional developers, but studies show that it has mixed effects in classrooms [15]. Thus, in Section 3, we propose a method for instructors to choose processes to include in classrooms and make adaptations to selected processes when necessary.

Besides selecting and adapting processes, student evaluation is also challenging in project-based courses. While other courses assess students with artifacts, such as code and report, project-based courses focus on *how* artifacts are achieved. Studies show that frequent feedback on student behaviors and processes are critical in project-based courses [2, 16], but they also show that providing such feedback is demanding. There are several challenges to process feedback. First, feedback is only useful when provided timely and continuously [17, 18]. Ideally, students receive feedback right after the process happens. In classrooms, however, instructors cannot observe all student behaviors and usually rely on artifacts generated by the process. Thus, instructors need time to analyze artifacts and provide feedback, and artifacts cannot provide full information about a process. Second, providing feedback demands expertise. On the one hand, Agile and XP methods are dynamic and flexible. Professional Agile teams commonly tailor their processes according to the team's situation [19]. Therefore, to understand *why* a process fails in a given student team is challenging, requiring expertise and efforts. On the other hand, although students work collaboratively in teams, individual evaluation is important from a pedagogical perspective [6, 7]. Education studies show that individual accountability is important in collaborative learning environments [20]. Accordingly, instructors need to carefully examine each student's behaviors and pedagogically design proper feedback for each student, which requires efforts and teaching expertise. Thus, in Section 4, we present a method for building automated process metrics to address the challenge.

This report is organized as follows: Section 2 discusses some related studies, focusing specifically on designing XP courses and building XP process metrics. Section 3 presents a research study on the first issue: what XP processes should we choose. Section 4 presents a research study on the second issue: how can we build XP process metrics? Section 5 presents some future research directions. Section 6 concludes this report.

2 Related Work

Teaching Agile methods is an important research topic and practical issue. There are several mainstream Agile methods [21], such as eXtreme Programming [5], Scrum [22], and Lean software development [23]. All these Agile variants emphasize flexibility so that software development teams can handle fast-changing environments. In this study, we focus on eXtreme Programming (XP) [4]. We do not argue that XP is superior to other Agile variants. Our choice is based on the research constraint that the software engineering course under examination teaches XP method. Besides, XP is widely used and shares many similarities with other Agile methods; thus, most of our findings are generalizable to other methods as well.

In this study, we focus on project-based courses. Meanwhile, we recognize that there are other teaching methods for Agile education. For example, LEGO is widely used by researchers and instructors to create a simulated and controlled learning environment [24, 25, 26]. However, these alternative methods can only provide simulated software development environments, while project-based courses, especially project-based courses with customers [27], can provide an environment where students work on real-world tasks. Thus, we believe project-based courses are and will continue to be an essential teaching method for software engineering education.

Team projects are complex. Previous studies have explored various design spaces, such as team formation [28], project selection [29], team coaching [30], and student evaluation [31], while others have reported experience regarding the design, organization, teaching, and evolution of a project-based software engineering curriculum [2, 6, 7, 32]. In this report, we focus on course design and student evaluation, as they are the bases of any course.

Before discussing related work on these two specific topics, we want to point out that our research focus is not evolving teaching methods *with* Agile or XP [33, 34]; instead, we examine teaching methods *for* XP.

2.1 Design XP courses

Several studies and experience reports have shown that the academic environment has constraints that prevent replicating industrial Agile processes in classrooms [32, 35, 36, 37, 38]. For example, it may be impossible for students to meet at a time other than scheduled lectures [38] while working together is crucial in XP for fast feedback and close collaboration [5]. As another example, a student's working time is usually unevenly distributed, and they have unpredictable schedule, which makes progress checking and effort estimation more difficult. Given these and other constraints, instructors should carefully determine what processes students should follow and make necessary adaptations [15].

Masood et al. [39] examined course constraints, adaptations that students made to Agile processes, and effects of those adaptations. They recommended all Agile processes to students and observed how students adapted processes due to course constraints. It was unclear, however, whether and how adaptations made by students organically influenced learning outcomes. In contrast, we proactively adapt processes based on industrial observations and a team process taxonomy. We believe this method leads to a course design that has a firm real-world grounding [1] and clear learning benefits designed by *instructors*.

Similarly, Goldman et al. presented an experience report teaching XP in classrooms [15]. The authors found that adaptations may be needed because not all XP practices are possible or desirable in all situations. They made "retroactive" recommendations based on their experiences. In comparison, instructors can use our method presented in Section 3 to make informed choices and adaptations without experimenting with real students, thus reducing the risk of undesired outcomes.

Another issue that may prevent students from learning Agile processes is that processes that benefit professional teams may not benefit student teams in the same way [40]. This issue is exaggerated by that Agile teams are self-organizing teams [36, 7]. Thus, besides selecting processes for students to use, we must take care to adapt them in a way that demonstrates substantial benefit to student teams in classrooms. We rely on the taxonomy and our field study for deciding how to adapt processes while preserving the essence of each process.

2.2 Process metrics in software engineering courses

We build process metrics for coaching student software engineering teams. Coaching is a decisive factor in software engineering education [41, 42]. Instructors should follow student teams carefully and provide timely guidance to students. Meerbaum-Salant et al. have presented a mentoring model [42], composed of 17 practices covering various aspects of a software engineering course. In this report, we improve coaching via automated process metrics. These metrics can support team coaches to provide feedback more efficiently and accurately. Besides, process metrics can be applied in parallel with any framework or mentoring model.

Some previous studies on software engineering education have reported the use of metrics. Matthies et al. have built ScrumLint [17], an automated system, in their course. Alperowitz et al. presented some metrics that they used for managing student projects [43]. These studies inspire out work. We will explain the importance of measuring *processes* and study the effectiveness of metrics systematically in a case study.

Besides metrics, researchers have examined evaluations in Agile courses more broadly. For example, Meier et al. provide a taxonomy called the Agile Competence Pyramid for understanding Agile requirements [44]. They argue that this model supports better evaluation of student teams by grouping Agile requirements into three categories. Nevertheless, they and other studies provide little discussion or research on metric design, and their evaluation of process metrics largely relies on human efforts. In comparison, our work presented in Section 4 combines the understanding of Agile requirements with the design of process metrics, and we examine designed metrics scientifically in a case study.

Metrics for software artifacts and processes have been studied in industry [45], including methods for measuring process conformance [46, 47]. Process metrics are used for various reasons, such as defect prediction [48]. We build metrics to teach XP processes, which is quite different from the goals of industrial metrics. Meanwhile, we borrow some industrial practices when designing metrics. For example, we use Zazworka et al.'s *Process Conformance Templates* to design and describe a metric [47], but unlike their use of Process Conformance Templates, we align each metric with expected learning outcomes.

Our process metrics are built with data from software development tools. These development tools are widely used in project-based learning courses, according to a recent survey [49]. For example, GitHub is used for various purposes in classrooms [50]. Thus, our metrics and method for building those metrics are applicable to modern software engineering courses. There are also studies on mining software development artifacts for industrial purposes [51]. Their goals of analysis include defect prediction [51], process conformance [46], and others. Data sources used in these works range from APIs of software development tools [52] to individual plugins that have to be installed on IDEs [53]. Our work focuses on metrics of process conformance based on data available from APIs of online software development tools, and our metrics are for teaching purposes.

3 What XP processes should we use in classrooms?

Instructors want to teach students *all* XP processes that they may use as professionals [2, 15]. However, with limited resources and various course constraints, instructors must decide *which* processes to include and *how* to adapt them for classrooms [54, 39, 15]. The decision should not be arbitrary; in this report, we choose as our main constraint that ***the selection of processes should be*** **complete** ***with respect to a well-established taxonomy.*** This section presents a method for instructors to make choices and adaptations informed by professional processes.

Instead of using experiments that may cause undesired outcomes, we rely on an established team process taxonomy of Marks et al. [18], which shows three categories of team processes essential for any working teams, including software development teams. With a field study, we combine the taxonomy with professional processes at a software company where a variety of Agile processes form part of the daily workflows. By mapping both professional processes and our current pedagogy into the taxonomy, we identify areas in our course design that are under-covered. To decide which additional processes to add so that our course is complete with respect to the taxonomy, we ask professionals to rate each process's importance in their daily workflow through a survey and use this data to prioritize filling the gaps. Then, we augment a course design with XP processes that are important for professionals and *complete* for working teams with respect to the taxonomy's main categories. With this method, we improve our course's *completeness* without exprimenting on real students.

We inform our selection of Agile processes using team process theories. Moe et al. [55] used a team process theory to observe a team transitioning to the Scrum methodology [2]. They were able to connect observed barriers to success with the theory, showing that team process theories could explain the failure and success of Agile teams. Similarly, Lindsjørn et al. [56] borrowed ideas from teamwork theories to examine the success factors in Agile software development. They designed a survey based on a teamwork theory to measure teamwork quality and team success. Based on the survey data, they showed that teamwork is a significant predictor of team performance. The two studies showed that team theories could provide useful insights and measurements for Agile teams. In this study, we choose Marks et al.'s team process taxonomy [18]. We recognize that there are other successful team process theories [57, 58, 59], but they do not contradict Marks et al.'s taxonomy. Instead, they present alternative perspectives that confirm and extend the taxonomy. Thus, this taxonomy can serve our goal of selecting processes that are *complete* for working teams, since the format of taxonomy gives us a straightforward way to define measurements for *completeness*.

We use a case study to examine two research questions about the augmented course design. First, as both students and instructors have limited resources, we investigate whether new processes interfere with learning opportunities for XP processes that are previously included in the course design. This undesired outcome may happen when experimenting a pack of new processes with students. Second, we ask whether the new processes deliver benefits to the students comparable to those experienced by professional teams, and whether students recognize the usefulness and value of the new processes. As studies have shown, students may fail to recognize the value of a process if it is improperly used in classrooms [38, 15]. Our case study shows positive results for both questions by comparing self-reported performance from this case study with a previous offering of the same course. These results suggest that our method could achieve desired learning objectives while avoiding undesired outcomes such as overloaded students.

Section 3.1 describes our field study of a successful, mature professional organization, explaining how XP processes map to the taxonomy and allowing us to choose three processes to fix gaps in our pedagogy. Section 3.2 describes the results of incorporating the three processes into the course, validating the benefits of the augmented course design.

3.1 Observe, Classify, and Choose XP Processes

Pivotal[1] is an American company that provides a range of services. Extreme Programming (XP) [5] is widely adopted at this research site. Pivotal Labs, a division of the company, helps its customers transition their organizations to Agile/XP while delivering software to those organizations using Agile/XP. Pivotal is also active in research collaboration. Previously, they have collaborated with other researchers on software engineering studies [60].

We observed 3 Pivotal teams of different maturity levels. Our goal was to observe processes that teams used in an effective workflow and classify those processes with a taxonomy. The eventual goal of doing so was to identify a subset of processes for incorporating into our XP course according to the following criteria:

- The process is judged important by professionals.
- The process fills a gap in our course design, so that student teams use processes that are *complete* for working teams with the augmented design.

[1]https://pivotal.io/

Category	Practice	Definition	Importance
Transition Process	Pre-IPM	A meeting prior to an IPM where stories are prepared.	1.8
	Iteration Planning Meeting (IPM)	Create an iteration plan [5, 22, 23].	2.8
Action Process	Backlog	Create, prioritize, implement, and accept/reject user stories [5, 22].	3.0
	Continuous Integration (CI)	Frequent integration of individual's code into a mainline [5].	3.4
	Information Radiator	Set up an information monitor in the working space to track key information [5].	2.1
	Pairing	Pair programming applied to coding, managing, and design works [5].	2.9
	Standup	Short daily meetings [22].	2.4
	Test-Driven Development (TDD)	write tests before writing code [5].	3.2
Interpersonal Process	**Retrospective**	Discuss team problems openly at the end of an iteration [22, 23].	3.5

Figure 1: Categories in the taxonomy, Agile practices, explanations, and their importance. We choose IPM, Retrospective, and Standup (all highlighted in the table) to examine in the case study. Importance of a practice is averaged over survey responses (0="Not at all important", 1="Slightly important", 2="Somewhat important", 3="Very important", 4="Extremely important").

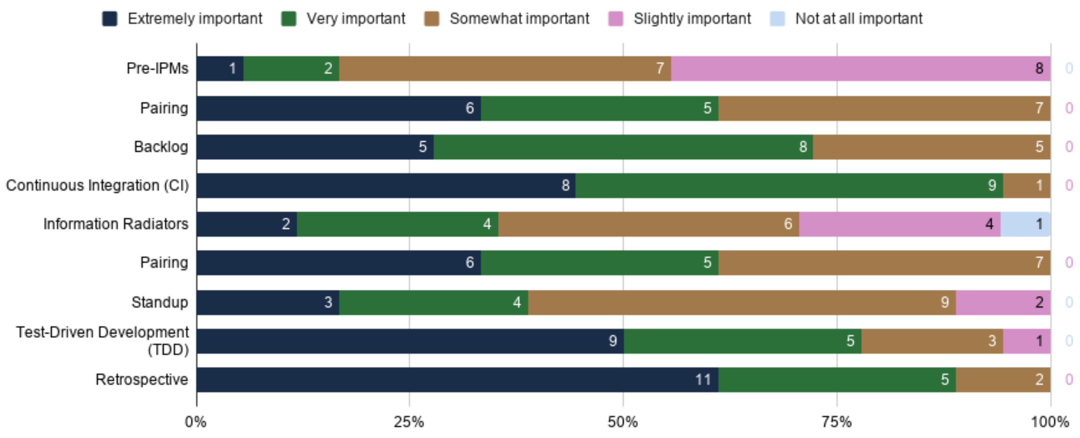


Figure 2: Breakdown of process importance measured by a survey.

3.1.1 Data collection

We use observations, interviews, and a survey to collect data. Data is collected between May 2018 and September 2018.

Observations. We conducted observations on 3 teams for about one week each. (The company uses 1-week XP iterations, so observing a team for one week allows us to observe all the processes they use during a typical iteration.) We also participated in a few meetings of other teams. In the observations, we started with a broad interest about workflows and focused on a subset of behaviors/meetings that were most relevant to the taxonomy (listed in Figure 1) as the research proceeded.

Interviews. For data triangulation, we conducted 5 semi-structured interviews with 4 employees (1 product manager, 3 engineers).

Survey. We designed a 12-question survey asking opinions about the importance of different processes within Pivotal[2]. A contact from Pivotal distributed the survey through an email list. We received 18 responses.

3.1.2 Analysis: Transition, Action, Interpersonal Processes

We use Marks et al.'s taxonomy of team processes [18] as the basis for our work; other theories, such as Salas et al.'s "big five" [57], could also be used. Future work may illuminate which theories or taxonomies are best used for adapting professional practices into an educational setting.

Marks et al.'s taxonomy of team processes [18] is developed from studies on general teams, and distinguishes *transition, action,* and *interpersonal* processes, among others:

- Transition processes are ones in which "teams focus primarily on evaluation and/or planning activities to guide their accomplishment of a team goal or objective." In our observations, we identified IPM (Iteration Planning Meetings) as transition processes. (Figure 1 explains all the processes described in the rest of the paper.)
- Action processes or phases are "periods of time when teams conduct activities leading directly to goal accomplishment." In our observations, we identified Backlog, Continuous Integration, Information Radiator, Pairing, Standup, and Test-Driven Development as action processes.
- Interpersonal processes "typically lay the foundation for the effectiveness of other processes." In our observations, we identified Retrospective as an interpersonal process.

We use a closed coding method to explain our observations, with codes derived from the taxonomy. Observation notes, interview notes, and transcripts are coded line-by-line. One researcher coded the data alone. Figure 1 lists all the processes we observed and which of the three categories they fall into. Processes with names **in bold** are the ones we decided to incorporate into our course, and which we describe in more detail in the rest of this section. In the next section, we justify the choice of these specific processes.

IPM is a transition processes.

An Iteration Planning Meeting (IPM) occurs at the beginning of each iteration. One objective of the IPM is to develop a shared understanding of the value, constraints, and acceptance criteria of each *user story*, the basic unit of work during an iteration, "to ensure everyone is on the same page and has a common understanding of the stories in the backlog" (as one survey response states). In the IPMs we observed, one person usually explained each user story, the team clarified each story's expectations through explanations and discussions, and the meeting would not proceed until all developers indicated that the expectations were clear. Some details of this discussion might be recorded in the story's description. IPM is directly linked to planning; therefore, it is a transition process.

Standup is an action process.

A primary objective of Standup is to discuss new findings and blocking factors. Standup is a short (a few minutes) daily meeting in which members take turns to update their status and plan. One survey response found Standup to be "most effective as a daily 'health check' and context-sharing activity". One interviewed engineer mentioned that "problems would be surfaced by a Standup and thus other team members can help to fix them". Standup directly contributes to the team's goal accomplishment by unblocking the team and sharing team status. Therefore, Standup is an action process.

Retrospective is an interpersonal process.

The Retrospective occurs at the end of each iteration, after the iteration's work has been delivered. During a Retrospective meeting, the team's practices are analyzed and improved. As one interviewed engineer put it, "[Retrospective] is the only practice that allows the team to tweak the practices" and "Retrospective is the place where you analyze your process and try to make it better". Furthermore, Retrospective regulates team emotions. In a typical retrospective, members express feelings on a message board, physical or virtual, with three columns for different feelings (a standard version is "happy, meh, and sad", where "meh" generally stands for a neutral feeling). Retrospective is thus expected to be a place where team members can safely share their feelings so that they would stay effective in other processes. One inteviewed engineer mentioned that "creating that space where everybody on the team feels like they can say things that they are feeling" is important for Retrospective. In summary, Retrospective lays the foundation for teams to stay productive, and thus, Retrospective is an interpersonal process.

[2]Survey can be found at https://github.com/ace-lab/xp-study-sigcse20/blob/master/PivotalProcessSurvey.pdf. In this survey, "sit together" is the Information Radiator process in Figure 1. We made this change for a better communication.

3.1.3 Select Processes

Figure 1 and Figure 2 shows the professional practitioners' survey responses on process importance. All processes, except for pre-IPM, are rated as "Somewhat important (2)" to "Extremely important (4)" on average.

In selecting processes, we first exclude those that are already covered in previous and current offerings of the course: *Backlog*, *CI*, *TDD*. (We describe the course more completely in Section 3.2.) We exclude *Information Radiator* and *Pre-IPM* because their importance as judged by professional practitioners is low, and because we do not want to overwhelm students with too many processes in a single course. While previous offerings of the course have encouraged (but not mandated) *Pairing*, and the professional practitioners we interviewed practice it almost exclusively, previous studies suggest that pairing may be impractical to do consistently in classrooms [40].

This leaves *IPM*, *Retrospective*, and *Standup* to focus on in our case study. We note that in addition to being missing from previous offerings of the course, these three processes provide students with learning opportunities in all three categories of Marks et al.'s taxonomy: as discussed above, *IPM* is a transitional process, *Standup* is an action process, and *Retrospective* is an interpersonal process. Therefore, adding these processes improves the *completeness* of the course from the perspective of covering the taxonomy.

3.2 Case Study

This case study was conducted at University of California, Berkeley from January through May 2019, in the context of CS169, a 14-week project-centric software engineering course that teaches XP. The course had 120 students divided into 20 teams. Each team worked with a customer to develop or enhance an existing Web application tailored to the customer's small-business need.

During an initial short iteration, student teams met with their customer to understand the requirements and goals for the project. The teams then worked on the project during four 2-week iterations.

The course was supported by two 20-hour-per-week teaching assistants, each of whom supervised and coached 10 student teams, meeting with each team during each iteration to provide feedback and evaluation. Both teaching assistants were also researchers in this study.

We compare this case study with the most recent previous offering of the same course. It had 126 students divided into 21 teams. The course had the same course project schedule and focused on XP as well. In this case study, we made the following major updates to the course design:

- Added checkpoints for IPM, Customer Meeting, and Retrospective (Section 3.2.1), in order to incorporate the processes.
- Updated the self-assessment survey (Section 3.2.2) to track newly incorporated processes.
- Gave students access to a new metrics dashboard that was different from the one used previously. We made this change migrating the dashboard to a new platform, and the main differences were on the dashboard's UI. The new dashboard presented a similar set of metrics about Backlog, CI, and TDD as the previous one. Thus, we expected this change to have a minimal impact on this case study.
- Instructors had access to data from newly added tools (Section 3.2.1). Although the data was not used for grading, instructors could use it to provide feedback in meetings.

The case study focuses on investigating whether students benefit from these augmentations, which were guided by the approach described in Section 3.1. We focus on the following two questions:

- Do new processes interfere with learning opportunities provided by processes already present in the course?
- Do students appreciate the usefulness and values of new processes for XP development?

3.2.1 Embed Processes

Figure 3 shows the iteration schedule. Each iteration is two weeks long, but we leave some overlap so that students have enough time for retrospectives. Thus, the start of week 3 is also the start of the next iteration.

Because teaching processes is demanding for instructors [2, 44, 7, 6], processes are supported by a student-facing tool that generates usage data that gives instructors some indirect visibility into how the team is using the process. Tools also allow students, who are new to the processes, to practice with more structured support. The tool for IPM is adapted

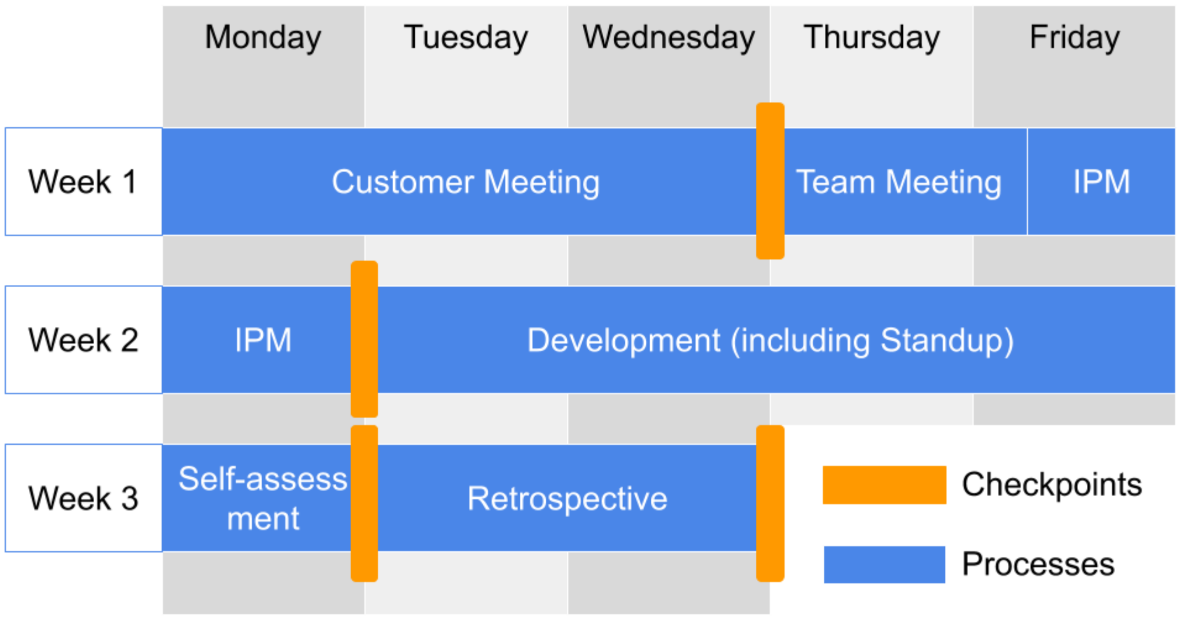


Figure 3: Iteration schedule. The development stage includes other processes such as CI, Backlog, and Standup.

from an open-source tool *planning poker*[3]. The tool for Retrospective is adapted from an open-source tool *postfacto*[4]. The tool for Standup is a Slack bot adapted from an open-source project *slack manager*[5].

Besides tools, we use explicit checkpoints to make processes transparent to students. Checkpoints are implemented as online surveys that students complete[6]. Checkpoints make instructor expectations explicit so that students can follow and improve their practices over time. In our case study, participation in these checkpoints is mandatory and graded. Each checkpoint contains questions that are related to the process. For example, the IPM checkpoint asks students to report whether they understood the goal of this iteration, among other questions.

For each iteration, we changed two or three questions for teaching purposes. For example, in Iteration 4 we added two questions each to the IPM and Retrospective checkpoints. One question asks how confident the student is in contributing to a productive process (IPM or Retrospective) in a professional team. The other asks how likely the student would be to use this process (IPM or Retrospective) in the future if the student were leading a team.

3.2.2 Data Collection

We collect data about tool use from checkpoints, and data about learning outcomes from checkpoints and self-assessment surveys[7].

The self-assessment survey contains both questions used in previous courses and questions newly added in this case study. For example, we add questions about team communication to measure the effect of Standup. The survey is adapted from surveys used and validated in previous studies, focusing on XP processes [61], software development waste [60], and teamwork [62].

We acknowledge that our data is not an objective measurement of learning outcomes. However, studies show that self-efficacy—students' belief and self-confidence in their capabilities and in their ability to use a particular course of action to achieve a desired result [63]—is a reasonable predictor of student performance [64]. Our self-assessment

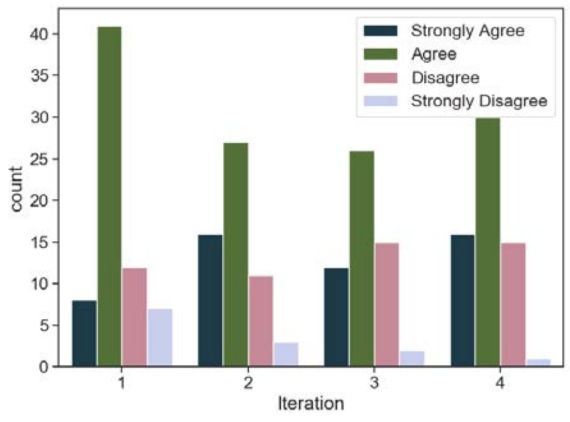
[3]https://github.com/bimovidia/planning-poker

[4]https://github.com/pivotal/postfacto

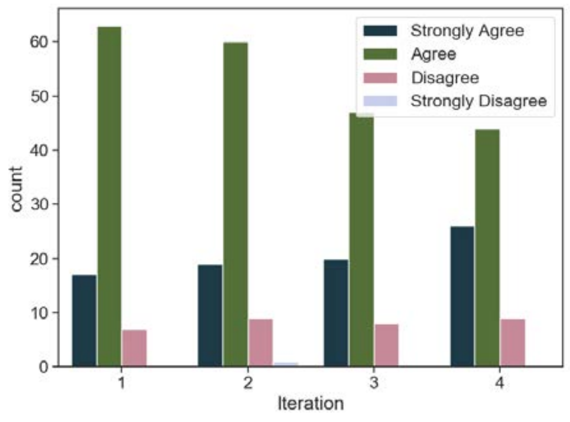
[5]https://github.com/anonrig/slack-manager

[6]IPM checkpoints can be found at https://github.com/ace-lab/xp-study-sigcse20/tree/master/IPM. Retrospective checkpoints can be found at https://github.com/ace-lab/xp-study-sigcse20/tree/master/Retrospective.

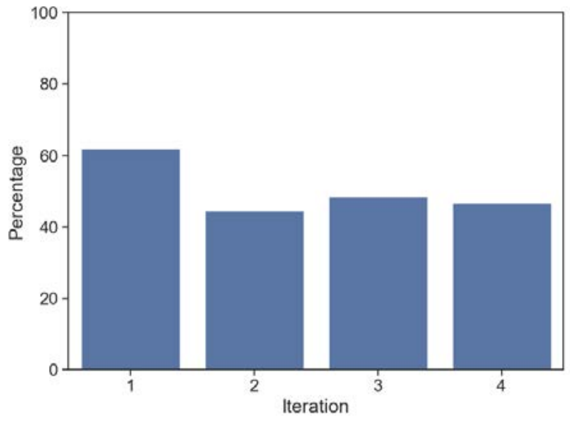
[7]Self-assessment surveys can be found at https://github.com/ace-lab/xp-study-sigcse20/tree/master/Self-assessment.



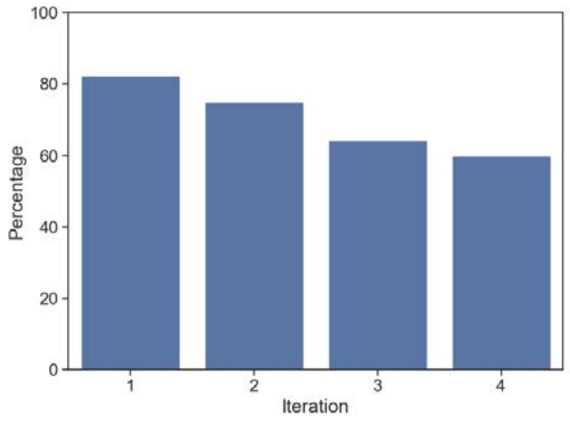
(a) Usefulness of *planning poker*



(b) Usefulness of *postfacto*



(c) Use of *planning poker*



(d) Use of *postfacto*

Figure 4: Tool use and tool satisfaction reported by students ($107 \leq N \leq 117$ for all iterations). Data for *slack manager* is not present because we do not have a checkpoint for Standup.

surveys and checkpoints measure self-efficacy, so we use them as reasonable proxy measurements for learning outcomes. Previous studies have also shown that surveys are useful instruments for teaching software engineering processes [17].

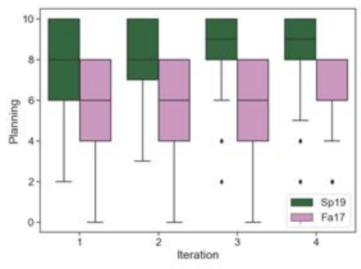
3.2.3 Results

Students followed the processes and were satisfied with the way the processes are supported.

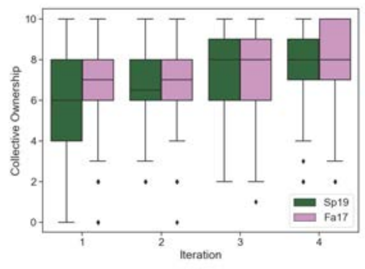
Student-reported tool usefulness and tool use are presented in Figure 4. Most students agree that the tools (*planning poker* and *postfacto*) are useful for their team development.

While students felt that the tools were useful, Figure 4c and Figure 4d show a low percentage of tool use (between 40%-60% for *planning poker* and 50%-80% for *postfacto*). Further investigations showed that in most cases, students used an alternative approach for the process. For example, students reported in Iteration 4 that they used in-person discussions instead of the tool. One student mentioned that "we implemented this into our general meeting pretty naturally, so we didn't really need it".

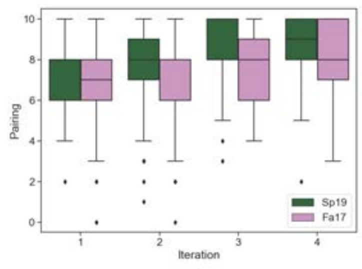
Results show that *postfacto* is more welcomed than *planning poker*. Based on survey feedback and team meetings, one reason is that *planning poker* is too "heavyweigh" for students. Compared with *postfacto*, which needs only one link to start a meeting, *planning poker* requires users to log in with their account. Furthermore, some students reported that they did not enough stories to discuss or their stories were too simple. Since *planning poker* had several steps to plan a story, in those cases, students found that the tool was too costly compared with an informal team discussion.



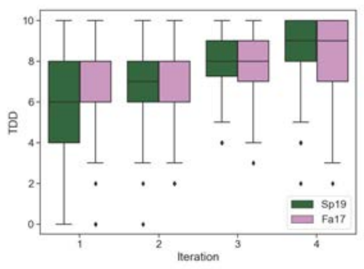
(a) Planning



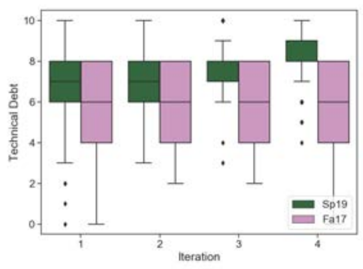
(b) Collective Ownership



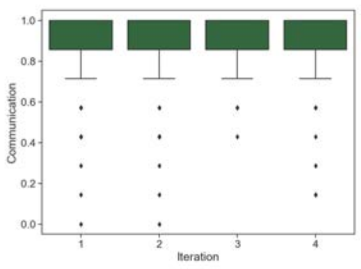
(c) Pairing



(d) Test-driven Development



(e) Technical Debt



(f) Communication

Figure 5: Self-reported process performance (communication is not previously measured). $112 \leq N \leq 115$ for all iterations.

Iteration 4's survey results suggested that students had proper training on IPM and Retrospective with the new course design. 41 (38%) students reported that they were "extremely confident" in contributing to a productive IPM in a professional team while 48 (45%) reported that they were "very confident"; 41 (37%) students reported that they were "extremely confident" in contributing to a productive retrospective meeting in a professional team while 51 (46%) reported that they were "very confident".

The inclusion of new processes does not interfere with existing learning opportunities.

In this case study, students achieved similar or better self-reported performance on all previously tracked XP processes. Presented in Figure 5a to Figure 5e, the median of self-reported performance on these tracked processes in this case study (Sp19) is the same or higher than the previous semester (Fa17) in Iteration 4. Furthermore, with an alternative hypothesis that Iteration 4's score in this case study is higher than the score of the previous semester, a Mann-Whitney rank test shows that student performance is significantly better on Planning, Pairing, and Technical Debt ($p \leq 0.01$). This result indicates that the augmented course design does not negatively influence existing processes.

Students appreciate the usefulness and value of the new processes (IPM, Retrospective, Standup) for XP development.

Planning improves significantly in this case study (Figure 5a). This is directly linked to IPM. When asked in Iteration 4 whether they would use IPM in the future if leading a team, 58 (54%) responded "definitely" and 41 (38%) responded "probably", which suggests that most students appreciate the usefulness of IPM for team development.

Improvements (value differences) measured by self-assessment surveys between two consecutive iterations are significantly higher in this case study than the previous semester for all processes except for *Pair programming* and *Planning* (Mann-Whitney rank test, $p \leq 0.05$). This observation indirectly supports the claim that Retrospective has changed student behaviors, since Retrospective focuses on process improvements. When asked in Iteration 4 whether they would use Retros if leading a team in the future, 51 (46%) responded "definitely" and 49 (44%) responded "probably", which suggests that most students appreciate the usefulness of Retrospective for team development.

Figure 5f shows that teams have a high[8] communication measurement across iterations. Communication is measured by survey questions adapted from a standard questionnaire for professional teams [62] and the score is averaged within each team. It indicates that teams have good communication, which is related to the use of Standup.

In summary, self-assessment surveys show improvements that are related to *IPM*, *Retrospective*, and *Standup*. It indicates that student teams benefit from the processes as professional teams. Therefore, students can appreciate the values of the processes, which is supported by Iteration 4's survey responses.

Summary.

When we changed our project-based course to incorporate *IPM*, *Standup*, and *Retrospective* selected in Section 3.1, we found that in comparison with a previous offering of the course lacking these processes: (1) students reported similar or better performance on existing XP processes, suggesting that our design augmentation does not interfere with existing processes; (2) student teams can benefit from new processes as professional teams, and students recognize the usefulness and value of these processes. Thus, we believe our method can generate augmented course designs that lead to better learning outcomes.

3.3 Threats to Validity

Field observations conducted in one company, however mature, do not necessarily represent the industry; other researchers can apply our methodology to observe other companies. Furthermore, we use our field study to emphasize the industrial grounding of our selection of processes, but practitioners may rely on secondary data sources for this purpose.

We rely on one case study to examine the new processes. In this case study, we did not control other counfounding factors that might influence student behaviors and self-assessments. However, we argue that it is difficult, if not impossible, to have a controlled A/B test in software engineering courses. First, we do not have enough resources to manage enough student teams for a statistically reliable comparison. Second, student teams work on different projects, which is a counfounding factor that we cannot avoid. Third, preventing some students from new processes that are considered important by professionals and are believed pedagogically helpful by instructors could propose an ethical issue. Thus, we choose case study as our research method.

This study focuses on XP. We do not argue that XP is representative of Agile or XP is superior to other Agile variants, such as Scrum [2]. We choose XP because the course examined in Section 3.2 focuses on XP. In the meantime, we believe our method is generalizable to other Agile variants, since all Agile methods stem from a similar set of values. In fact, *IPM*, *Retrospective*, and *Standup* are also important processes in Scrum. We encourage more studies to be done on other Agile variants.

4 Building process metrics in classrooms.

After XP processes are properly selected, it is essential to build proper measurements for those processes. Measurements are essential for software engineering [45] and teaching software processes [65]. They provide insights into student behaviors, and thus allow for better evaluation and reflection.

However, team-based software project courses are known to be instructor-intensive, especially because of the need to measure and give feedback on student team processes [6]. In recognition of this problem, we build automated metrics to support the measurement of XP processes and provide more insights for instructors.

In computing education, software metrics developed in industry can be readily deployed in classrooms as measurements. Although some metrics help novice programmers [66], they are less successful in software engineering projects [67]. Valued outcomes of an industry project, such as budgets, schedules, and business values [68], are often less important in a classroom setting. Course designers care not only about successful student "products" but also about *why* a student or team's artifact is of high or low quality [69]. Thus processes, not usually considered as essential measurements for project success in industries, are critical to successful learning.

In this report, we present a method to design measurements for processes in a software engineering course to facilitate learning. In particular, we establish the validity and reliability of designed process measurements built on data available from online software engineering development tools. Our experience with a prototype system supports the importance of processes in software engineering courses and the validity and reliability of our process metrics.

[8]In comparison, professional software teams using the same set of questions report an average of 0.84 (normalized) on communication [62].

4.1 Processes Are Key to Agile Software Projects

A successful XP team is one in which both the artifact (codebase) and the team are well-positioned to continuously maintain and evolve the software, adapting and refining it in response to changing customer requirements, environmental constraints, and other factors. For two reasons, we assert that evaluating XP teams in a classroom setting requires measuring how well the teams perform *processes*.

First, measuring outcomes of a software project rarely provides actionable information on how to improve those outcomes. For example, several static-analysis metrics that are widely accepted as proxies for software maintainability can be measured directly. But coaches (instructors, teaching assistants, etc.) in classrooms would like to know whether the students were following *processes* that are generally believed to promote maintainable code, especially if the team's codebase scores poorly on these metrics.

The second reason to measure processes is that some XP outcomes *cannot* be measured from the artifact alone. For example, a team is said to exhibit *collective ownership* when all team members are broadly familiar with the project's code, as opposed to each member being a specialist on their own code but unable to contribute in other areas. Collective ownership cannot be inferred solely from the codebase, as a team may have a beautiful codebase in which nobody is familiar with any code but their own. Teaching collective ownership requires observing the development process.

In principle, detailed measurement of a process requires constant observation of each team member's behaviors by the coach. Since that can be impractical due to limited resources (time, TA working hours, restricted communication channel, etc.) in classrooms, we propose to measure process conformance by formulating metrics computed from raw data obtained from collaboration tools the team uses. We refer to such raw data as *teamwork telemetry*.

We make two conjectures regarding processes and outcomes in an educational setting. Since one goal is to provide both students and coaches with actionable suggestions for producing successful outcomes, our first conjecture is:

C1. For an outcome of interest, teams that more frequently *conform* to the process(es) promoting the outcome will score better on the outcome, whereas teams that more frequently *violate* the process(es) will score worse.

A corollary to **C1** is that when a team is getting into trouble, we have an early warning of how and why they are doing so.

Furthermore, since in some cases we cannot measure processes directly, we make a second conjecture based on the methodology of Zazworka et al. [47]. We formulate process conformance regarding a *conformance template*, which defines observable metrics indicative of the process and acceptable thresholds on those metrics:

C2. We can define one or more metrics based on teamwork telemetry such that conformant behaviors relative to thresholds on those metrics imply that the process is being followed, and vice versa.

Outcomes	**Processes**	**Metric**
Collective Ownership	Using pull requests as code reviews will **promote** collective ownership.	m_1: Percentage of commented pull requests.
		m_2: Percentage of reviewed pull requests.
	Contributing to multiple layers (client, controller, etc.) will **promote** collective ownership.	m_3: Variance of number of edits in different layers.
	Having large, complex, frequently-changing files with a single contributor will **thwart** collective onwership.	m_4: Maximum percentage of edits for a file across all authors.
Programmer's Confidence	Following test-driven development will **promote** programmer's confidence.	m_5: Number of times the current test coverage is lower than the most-recently-measured coverage.
		m_6: Total number of edits to files containing test code.

Figure 6: Summary of outcomes, processes, and metrics mentioned in Section 4.

We next show two examples supporting these conjectures. Figure 6 gives a summary of outcomes, processes, and metrics covered in the rest of this section.

Topic	Content
Collective Ownership	People can change each other's code. People know how each part of the system is designed and implemented. 10 We know each other's work very well. Anyone can change code in any area. 8 We regularly share knowledge of codebase via meetings and code review. 6 Approximately, for each part of the codebase, more than half of our team members know how it is designed and implemented. 4 Approximately, for each part of the codebase, more than one of our team members know how it is designed and implemented. 2 Someone in our group can explain to us how each part is implemented. 0 I don't even know what I did in the past iteration!
Test-Driven Development	Do you have good test cases and automated drivers? 10 We always keep high test coverage. We always write tests first, using mocks and stubs as needed. 8 We keep good test coverage. We write tests first most of the time. 6 We keep good test coverage. But sometimes our test cases are designed specifically to cover certain parts of the code, instead of based on features/functions to be implemented. 4 We have moderate test coverage. We think writing tests adds too much overhead. We don't write tests for features/functions that are simple and straightforward. 2 We don't use automated testing tools. We test our system manually before delivering to the customer. 0 We don't really have any testing. Customers let us know if there are any problems though.
Programmer's Confidence	Do you think your team did something even you knew it was wrong? 4 We did our best. We would do the same thing given another chance. 2 We could have done better had we planned more carefully or communicated more efficiently. 0 Everyone knew that we are doing some things wrong. But no one bothered to point it out and make the change.

Figure 7: Following the format and methodology suggested by [61], students were given a prompt and asked for an integer response (0–10) rating their subjective impression of their team's performance on each topic.

4.2 Measuring Processes: Two Examples

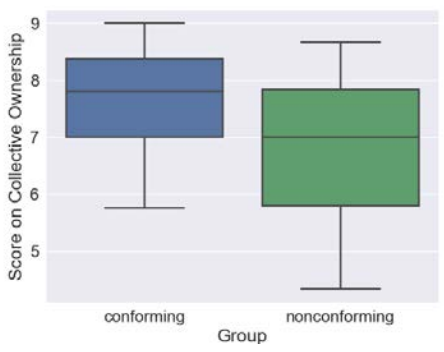
We have built a prototype system for student teams and instructors. The system tracks how well the team is doing on some processes and outcomes. The tool gathers teamwork telemetry for an arbitrary number of team-based student projects (we typically have 20 to 40 per course offering) using the application programming interfaces provided by team collaboration tools such as the code-hosting and team-collaboration tool GitHub, the project-management tool Pivotal Tracker, and the code quality measurement service CodeClimate. From the teamwork telemetry, our system computes and graphically presents various metrics of interest. An instructor can track team performance and give feedback using the system.

We deployed the system at University of California, Berkeley from August 2017 through December 2017, in the context of CS169, which is the same course described in Section 3.2. In this offering of CS169, there are 123 students, grouped into 21 teams (19 teams of 6, 1 team of 5, 1 team of 4). Similar to the course described in Section 3.2, each team builds or enhances a Web application in close collaboration with their customer, typically a nonprofit organization or campus unit. The team project consists of four 2-week iterations. Students individually complete self-assessment surveys (based on [61, 60], and graded only on participation) at the end of each iteration. Figure 7 lists some survey questions used in this offering of the course.

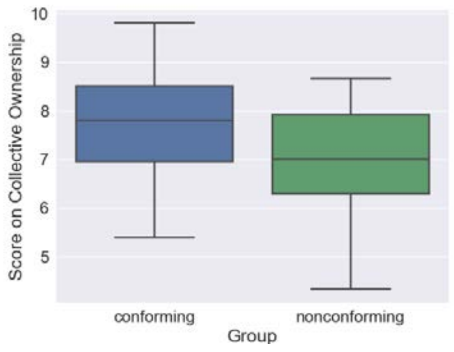
We will show two cases of measuring processes. In the first case, we identify three processes that promote collective ownership (CO) and develop metrics for evaluating whether a team is conforming to these processes. Our focus in this case is to investigate **C1**. The second case focuses on test-driven development (TDD), a process in which tests are written before code and help drive the design of the code. While following TDD is believed to result in higher programmer confidence in their team and codebase [70], our focus in this case is to support **C2**. Figure 6 gives a summary of outcomes, processes, and metrics described in the two cases.

4.2.1 Direct Observation of Processes for Collective Ownership

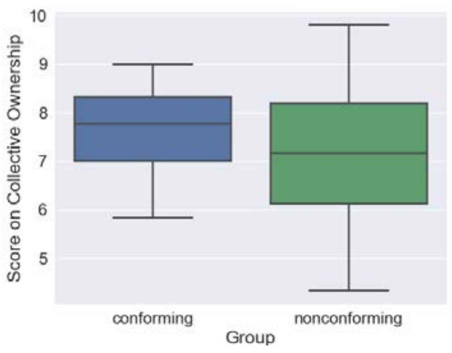
Recall that in a team with high CO (desirable), every team member is broadly familiar with the code written by other team members. Direct measurement of CO might involve individually testing each team member's ability to comment on or make changes to code they did not write, but this approach scales poorly. The outcome of CO reported in this



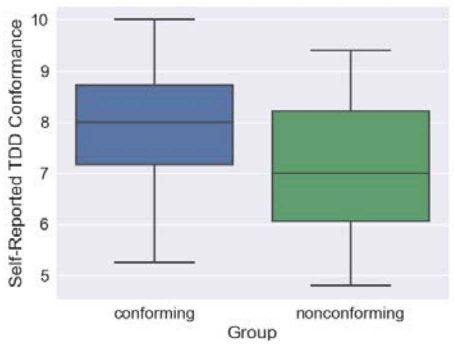
(a) **H1.1** Teams have higher percent of reviewed pull requests also have higher self-reported score on collective ownership.*



(b) **H1.2** Teams with more evenly distributed files edits on different layers have higher self-reported score on collective ownership.*



(c) **H1.3** Teams with more even distribution of authorship have higher self-reported score on collective ownership.



(d) **H2.1** Teams with nondecreasing test coverage and frequent test-editing activity have higher self-reported scores on conformance to TDD.*

Figure 8: For each hypothesis, we label measurements of team behavior at each point in time as *conforming* or *nonconforming*, and ask whether *conforming* group for a particular process has higher self-reported scores on the associated process or outcome. All except figure *c* (**H1.3**) are statistically significant ($\alpha = 0.05$).

section is measured from self-assessment surveys. We have identified two processes that promote CO and one process that thwarts it, and we state a hypothesis regarding the role of each process:

Promotes: Pull requests as code reviews. When one team member is ready to merge their code into the codebase, GitHub provides a mechanism called a *pull request* (PR) in which other team members have the opportunity to examine and comment on the proposed new code before it is added to the codebase. Having team members review each PR promotes CO.

H1.1 Teams that have a higher percentage of well-reviewed pull requests will have a higher self-reported score on CO.

Promotes: Contributions in multiple layers. Some software systems decompose naturally into subsystems/layers that call for different coding skills. In our study, Web services include client code (HTML and JavaScript), controller code (handles user interactions), model code (handles database queries), and more. Having each team member contribute code to multiple layers promotes CO.

H1.2 Teams in which members' contributions to the codebase are more evenly distributed over different layers will have a higher self-reported score on CO.

Thwarts: Large file with a single contributor. A large file that changes frequently (unlike, for example, configuration files, which rarely change once created), but whose changes are dominated by a single team member, thwarts CO.

H1.3 Teams with fewer large files dominated by one team member will have a higher self-reported score on CO.

The quantitative data to support the three hypotheses is available directly as teamwork telemetry. We evaluate the hypotheses by deriving metrics based on this teamwork telemetry and comparing the metric values with teams' self-assessment of CO.

Higher PR review rate leads to higher CO

We measure whether pull requests are reviewed directly from GitHub. For each team's codebase, we define m_1 as the percentage of pull requests with at least one comment:

$$m_1 := \frac{n_{\text{commented}}}{n_{\text{pr}}}$$

where $n_{\text{commented}}$ is the number of pull requests with at least one comment, and n_{pr} is the total number of pull requests.

However, not every comment constitutes a "review" of the code, so we filter out reviewers' comments (collaborators can be assigned as a reviewer for a pull request on GitHub) and build m_2 as

$$m_2 := \frac{n_{\text{reviewed}}}{n_{\text{pr}}}$$

where n_{reviewed} is the number of pull requests with at least one comment from reviewers.

To see if **H1.1** is supported, we distinguish N data points, each of which is a tuple of a team and an iteration. In this case, $N = 47$. Some data points are removed because the team does not have a valid score in that iteration. We label point i $(1 \leq i \leq N)$ as *conforming* if

$$m_1^{(i)} > \text{median}\left\{m_1^{(1)}, \ldots, m_1^{(N)}\right\} \textit{ or } m_2^{(i)} > \text{median}\left\{m_2^{(1)}, \ldots, m_2^{(N)}\right\}$$

and *nonconforming* otherwise. Figure 8a compares these labels with the average self-reported scores on CO from self-assessment surveys. A Mann-Whitney test confirms **H1.1** (Figure 9 shows the test results and interpretations for all hypotheses we examine).

More even distribution of file edits across application layers leads to higher CO

We measure the distribution of file edits directly from GitHub data. A file's layer is inferred with high confidence based on the framework's file naming conventions.

If we define $e_{i,j}$ as the number of edits by team member i to files in layer j, we can compute the average of variances of the number of edits across different layers over all team members:

$$m_3 := \text{Average}\left\{\text{Var}\left\{e_{i,j} \mid \forall j \in \mathit{layers}\right\} \mid \forall i \in \mathit{members}\right\}$$

We categorize each data point ($N = 63$) as *conforming* if

$$m_3^{(i)} < \text{median}\left\{m_3^{(1)}, \ldots, m_3^{(N)}\right\}$$

and *nonconforming* otherwise. Figures 8b and 9 show self-reported conformance of CO is significantly greater for the *conforming* group than the *nonconforming* group, confirming **H1.2**.

Fewer files dominated by one author leads to better CO

GitHub records each editing act by a team member on a file. If $p_{k,i}$ is the percentage of edits of team member i on file k, m_4 is defined as the average of the maximal percentage of edits across all members over all files:

$$m_4 := \text{Average}\left\{\max\left\{p_{k,i} \mid \forall i \in \text{members}\right\} \mid \forall k \in \text{files}\right\}$$

where *files* is the same set of files used in computing m_3, and the first commit to each file is excluded.[9]

[9]For framework-based applications, the first commit of a file often corresponds to boilerplate "scaffolding" provided by the framework itself and not attributable to any team member.

A data point i is *conforming* if

$$m_4^{(i)} < \text{median}\left\{m_4^{(1)}, \ldots, m_4^{(N)}\right\}, N = 63$$

Figure 8a shows that self-reported conformance to CO is greater for the *conforming* group than the *nonconforming* group. However, the difference is not statistically significant (Figure 9), so **H1.3** cannot be confirmed.

4.2.2 Indirect Observation of the TDD Process

Test-driven development (TDD) is a well-accepted software engineering process that can decrease defect rates [71] and increase code quality [72]. In ideal TDD, before writing a given piece of code, the developer writes a test for it, which obviously fails, then writes just enough code to make that test pass, then continues with further test cases until the behavior of the new code has been fully specified.

Following Beck [70], we hypothesize that following TDD leads to higher programmer confidence in both the codebase and the development process. Within self-assessment surveys, we observe a moderate correlation (Pearson's $r = 0.65, p < 0.01$) between self-reported TDD conformance and self-reported confidence, which confirms the hypothesis. In this case, we investigate: can we reliably detect whether a team is following TDD despite being unable to observe the process through direct measurement?

We measure two aspects of TDD: 1) tests should drive the development, and 2) test cases should be non-trivial. Derivatives of test coverage (the difference between current test coverage and the most-recently-measured coverage) is a good proxy for the first aspect, as a negative derivative implies newly added code without coverage of any test case, which indicates a violation of TDD. Assessing whether test cases are non-trivial is more difficult. Ideally we would use techniques such as mutation testing [73]. Here, we use the number of edits to files containing test code as a proxy. We assume that more edits in test files lead to test cases that are non-trivial.

Therefore we state:

H2.1 Teams whose test coverage over time is monotonically nondecreasing and who frequently edit files dedicated to test code will score higher on the self-reported conformance of TDD.

Let $0 \leq c_t \leq 1$ be the test coverage at time t, obtainable via the API of continuous-integration systems such as Travis[10]. We retrieve a sample of c_t every six hours for each project's codebase.

We calculate how often the test coverage *decreases* relative to its previous sampled value:

$$m_5 := \text{Count}\left\{t | c_t - c_{t-1} < 0, t = 1, \ldots, T\right\}$$

Now let *tests* be the set of test files and e_k the total number of edits to file k in the past seven days. We also calculate, over all commits in the past seven days, the average of the total number of edits to files containing test code:

$$m_6 := \text{Average}\left\{\text{Sum}\left\{e_{k,t} \mid \forall k \in \text{tests}\right\} \mid t = 1, \ldots, T\right\}$$

A data point is *conforming* (Figure 8d) if

$$m_5^{(i)} = 0 \text{ or } m_6^{(i)} > \text{median}\left\{m_6^{(1)}, \ldots, m_6^{(N)}\right\}$$

where $N = 62$.

We compare the self-assessment survey scores for the two groups. Figure 8d and 9 show that **H2.1** is confirmed.

4.3 Threats to Validity

Our work only establishes the correlation between processes and outcomes. The causality, although important to our approach, requires a more controlled experiment. Classrooms have various confounds that can influence the analysis. Students are not static. They learn new methods and receive feedback from coaches to improve their development skills over time. How the influence of these confounds can be minimized or removed, is a question that has to be answered by further analysis.

[10] In practice, we use the API of `codeclimate.com`, which in turn can retrieve coverage information from `travis-ci.org`.

Hypothesis	**Median**		**Sample Size**		**Results**	
	G_c	G_n	G_c	G_n	U	p
H1.1	7.8	7.0	30	17	352.5	0.016*
H1.2	7.8	7.0	32	31	647.5	0.019*
H1.3	7.8	7.2	32	31	592.0	0.094
H2.1	8.0	7.0	50	12	401.0	0.037*

Figure 9: Mann-Whitney test results with alternative hypothesis being that compared value is *greater* for G_c (*conforming* group) than for G_n (*nonconforming* group). (*) indicates the result is significant ($\alpha = 0.05$).

In our study, measurements of outcomes and conformance to some processes are based on self-assessment surveys. Although surveys are only graded on participation, the validity and reliability of survey results can potentially undermine the robustness of our analysis. Objective measurements of outcomes and conformance to processes are generally difficult to achieve. Some outcomes are intrinsically subjective, such as collective ownership reported in our study. Some processes are broad and multi-dimensional, such as test-driven development reported in our study. In what scenarios can these elements be measured and how, is a question that we plan to answer with future works.

Students (and less-experienced developers) make procedural mistakes that can confound our metrics. For example, we attempted to measure refactoring behaviors of student teams by (among other things) asking students to follow a specific format for commit messages that would allow automatic parsing of those messages to determine which commits might be related to refactoring. Since few students used our template correctly, we were unable to gather reliable data. We conclude that we must base our metrics only on simple assumptions and behaviors; for example, the metric of distribution of file edits only assumes the usage of GitHub, which is easy to enforce.

As another example, one student added a third-party JavaScript library into their team's repository and incorrectly labeled it as having been produced by the team, which led to a decrease in test coverage and undermined the validity of the TDD measurement. For the moment, therefore, we rely on our metrics and measurements more as an *early warning system* that a team may be behaving in a manner that could lead to poor outcomes. In the future, with more data and better models, we believe it is possible to detect these abnormalities and provide more accurate profiles of team status.

5 Future Work

Project-based software engineering courses have many design spaces that worth exploring. One such design space is real-world collaborations. Our experience suggests that students' learning experiences are highly correlated with their projects. In CS169, the course under examination in this report, students work with real-world customers. This method is also used in other software engineering curricula [27]. As customer's projects are profoundly different, ranging from websites for small businesses to API services for large organizations, students are motivated differently, work on different tasks, and thus have different learning experiences. We have seen some pioneer studies in this design space, such as [27, 74, 75]. Nevertheless, there are no commonly accepted "best practices" for vetting, interacting, and managing customers. For example, how can we manage customer relationships and products so that future students from the same course could maintain the same project? We believe a systematic understanding of real-world collaborations, one that shows the benefits and pitfalls of having real-world collaborations in software engineering courses from a dynamic and multi-session view, would have both academic and practical benefits.

While recognizing that software engineering courses are complex, in the rest of this section, we will focus on some extensions to our work on choosing XP processes and building XP process metrics.

5.1 Scaffolding systems for XP processes

Students are new to XP processes, and they may misuse a process. Anecdotally, we have found that teams with a student who had experience (from a previous internship, for example) with a specific process would lead to a better learning outcome for all other students in the same team. We hypothesize that this is because that team can use this process "correctly", leading to a better understanding of the process's role and values for software development. Accordingly, we believe it is necessary and useful to build systems that help students to use a process for learning.

In Section 3's user study, we give students access to some tools adapted from open-source tools, such as *planning poker* and *postfacto*. These tools are designed for professional teams, and thus they assume that users are proficient or familiar with the process. The assumed proficiency may cause problems for students. For example, we have observed that in some cases, students skipped team discussion in IPMs; instead, they assigned the story to a member and let that member decide the point. While we do not currently track this behavior, one can imagine that this anti-pattern could be detected indirectly from system behavioral data, such as the number of votes per story. Thus, we believe it is an interesting research direction to extend those tools and build metrics with tool data.

Furthermore, researchers should examine software development tools from a teaching perspective. It is possible to add mechanisms to those tools for students to use the process in a more informed way. In the case study presented in Section 3, we use checkpoints to explain a "standard" way of using a process. However, a preferred way might be to integrate instructions into those tools.

Although many extensions could be integrated into those tools, we caution that our case study suggests that students prefer simple tools. In general, tools and processes are useful for students when tasks are complex, but some tasks undertaken by student teams may be too simple to merit tool support, in contrast to the analogous tasks performed by professional developers. For example, in our case study, one student reported that "we didn't use [planning poker] because we had two very simple stories," so the team achieved consensus on effort estimation without the scaffolding provided by that tool. Therefore, while more instructions could be integrated, those tools should be kept as simple and lightweight as possible to accommodate these simpler tasks as well as students' lower time commitment to the course. This observation is consistent with Erdogmus & Peraire's recommendations in their experience report [6] implementing a flipped classroom for a software engineering course.

5.2 Using process metrics with more flexibility

XP process metrics presented in Section 4 are used in our case study as "standalone" metrics. Future studies could examine ways that use those metrics more flexibly.

"Standalone" metrics have limitations. They cannot capture the full picture, and they are influenced by some edge cases. For example, our metric using the distribution of file edits implicitly assumes that one commit has only one author, and one team member has only one username. However, these assumptions may break when the team uses pair programming or when a developer has different configurations on different development environments. Thus, a robust evaluation should rely on multiple data sources and metrics. We present in this part two potential solutions.

First, if we view XP process metrics as learning analytics, we can build visualizations and dashboards that leverage human intelligence to address the shortcomings of those metrics. Such a system is what Baker called "stupid" tutoring systems [76]. However, simply putting all metrics into one dashboard may overwhelm instructors, causing no benefit or even leading to misunderstandings. One of the preferred presentations is data storytelling [77], where metrics are organized into high-level information with various techniques. Future studies are needed on designing and automating the data synthesizing and visualization in order to present XP process metrics as data storytelling.

Second, XP process metrics could be combined with knowledge tracing models to measure processes systematically. Knowledge tracing models such as Bayesian Knowledge Tracing [78] and Deep Knowledge Tracing [79, 80] are widely used in tutoring systems. They are a principled way of synthesizing student performance measurements and provide an inference of the strength of knowledge components. Most knowledge tracing models are applied to teaching concepts such as math. In our XP courses, we can see an analogy where students' understanding and learning of an XP process could be inferred from XP process metrics. Compared with standalone metrics, knowledge tracing models can synthesize data from multiple sources in various formats. Besides, knowledge tracing models learn the proper weight of each process metric in measuring a knowledge component and consider inter-metric correlations such as temporal correlations. Thus, knowledge tracing models or other principled models could be an effective way of synthesizing XP process metrics.

6 Conclusions

Project-based software engineering courses are widely used for Agile education. In this report, we have presented a method for choosing a *complete* set of XP processes and a method for designing XP process metrics.

In the first part of this report, we selected three processes to augment a course design by combining Marks et al.'s taxonomy with a field study in industry, and we showed evidence that the augmented design improves learning outcomes in a case study without interfering with prior processes. With checkpoints and tools, students can benefit from the newly added processes the same way as professional teams, which suggests improved learning outcomes.

Teaching a *complete* yet reduced set of XP processes is essential, considering course constraints and limited resources. Our method presented in Section 3 helps instructors to budget their course resources while providing students with learning opportunities that are *complete* for professional careers. Course designers should reexamine their courses with our method. They can creatively adapt and tailor XP processes to fit course constraints. In our study, our method not only determines the *completeness* of a course design, but also captures the essence of various XP processes for adaptations. For example, we did not require product managers to lead a discussion at the IPM before estimating stories, although this was a common practice in Pivotal. This adaptation is justified because it does not influence the IPM's position in the taxonomy. Besides, we made a change to the frequency of Standups similar to Masood et al. [39], but unlike Masood et al., this change is informed proactively by Standup's role in the taxonomy.

In the second part of this report, we presented the importance of measuring processes in classroom software projects and a method for designing and implementing XP process metrics with data from online software development tools. Our method can facilitate systematic, targeted, and automatic evaluation of student projects in software engineering courses. Our XP process metrics can reduce the amount of effort an instructor has to spend on tracking and understanding student behaviors in teams, and thus increase the efficiency, scalability, and quality of project-based learning for software engineering education.

In general, teaching and measuring team processes is difficult [17, 6]. Our XP process metrics, together with software development and team process tools that generate instructor-visible data, as we have done in Section 3, can provide better insights into student behaviors. In this report, our XP process metrics do not use tools that support the three processes presented in Section 3, but in the future, we hope to use data from those tools to determine whether teams that adhere more closely to the processes experience better learning outcomes, produce higher quality software artifacts, or both.

Acknowledgement

This report is done in collaboration with Adnan Hemani, Joshua Zeitsoff, and Professor Yannis Dimitriadis. We also thank Anirudh Kothapalli, Anish Saha, and Xiao Fu for their help. Thank Professor Joshua Hug for his feedback on this report. Thank my advisor, Professor Armando Fox, for his time and effort coaching and advising me.

References

[1] Nancy R Mead. Software engineering education: How far we've come and how far we have to go. *Journal of Systems and Software*, 82(4):571–575, 2009.

[2] Andreas Scharf and Andreas Koch. Scrum in a software engineering course: An in-depth praxis report. In *Software Engineering Education and Training (CSEE&T), 2013 IEEE 26th Conference on*, pages 159–168. IEEE, 2013.

[3] M. Ardis, D. Budgen, G. W. Hislop, J. Offutt, M. Sebern, and W. Visser. Se 2014: Curriculum guidelines for undergraduate degree programs in software engineering. *Computer*, 48(11):106–109, Nov. 2015.

[4] Armando Fox and David Patterson. *Engineering Software as a Service: An Agile Approach Using Cloud Computing*. Strawberry Canyon LLC, San Francisco, CA, 1st edition, 2014.

[5] Kent Beck and Erich Gamma. *Extreme programming explained: Embrace change*. Addison-Wesley Professional, 2000.

[6] Hakan Erdogmus and Cécile Péraire. Flipping a graduate-level software engineering foundations course. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*. IEEE, may 2017.

[7] Nicole Herbert. Reflections on 17 years of ICT capstone project coordination. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education - SIGCSE 18*. ACM Press, 2018.

[8] Andrew Begel and Beth Simon. Novice software developers, all over again. In *Proceedings of the fourth international workshop on computing education research*, pages 3–14. ACM, 2008.

[9] Alex Radermacher and Gursimran Walia. Gaps between industry expectations and the abilities of graduates. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 525–530. ACM, 2013.

[10] Timothy C Lethbridge. What knowledge is important to a software professional? *Computer*, 33(5):44–50, 2000.

[11] Leigh Ann Sudol and Ciera Jaspan. Analyzing the strength of undergraduate misconceptions about software engineering. In *Proceedings of the Sixth international workshop on Computing education research*, pages 31–40. ACM, 2010.

[12] Viljan Mahnič. Scrum in software engineering courses: an outline of the literature. *Global Journal of Engineering Education*, 17(2):77–83, 2015.

[13] Bernd Bruegge, Stephan Krusche, and Lukas Alperowitz. Software engineering project courses with industrial clients. *ACM Transactions on Computing Education (TOCE)*, 15(4):17, 2015.

[14] R Keith Sawyer. *The Cambridge handbook of the learning sciences*. Cambridge University Press, 2005.

[15] Alfredo Goldman, Fabio Kon, Paulo JS Silva, and Joseph W Yoder. Being extreme in the classroom: Experiences teaching xp. *Journal of the Brazilian Computer Society*, 10(2):4–20, 2004.

[16] Vincent Leilde and Vincent Ribaud. Does process assessment drive process learning? the case of a bachelor capstone project. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, pages 197–201. IEEE, 2017.

[17] Christoph Matthies, Thomas Kowark, Keven Richly, Matthias Uflacker, and Hasso Plattner. How surveys, tutors, and software help to assess scrum adoption in a classroom software engineering project. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE 16*. ACM Press, 2016.

[18] Michelle A. Marks, John E. Mathieu, and Stephen J. Zaccaro. A temporally based framework and taxonomy of team processes. *The Academy of Management Review*, 26(3):356, jul 2001.

[19] Amadeu Silveira Campanelli and Fernando Silva Parreiras. Agile methods tailoring–a systematic literature review. *Journal of Systems and Software*, 110:85–100, 2015.

[20] Marjan Laal, Loabat Geranpaye, and Mahrokh Daemi. Individual accountability in collaborative learning. *Procedia-Social and Behavioral Sciences*, 93:286–289, 2013.

[21] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[22] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.

[23] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit: An Agile Toolkit*. Addison-Wesley, 2003.

[24] Maria Paasivaara, Ville Heikkilä, Casper Lassenius, and Towo Toivola. Teaching students scrum using lego blocks. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 382–391. ACM, 2014.

[25] Linda Laird and Ye Yang. Engaging software estimation education using legos: A case study. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 511–517. ACM, 2016.

[26] Stan Kurkovsky. Teaching software engineering with lego serious play. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, pages 213–218. ACM, 2015.

[27] Jari Vanhanen, Timo OA Lehtinen, and Casper Lassenius. Teaching real-world software engineering through a capstone project course with industrial customers. In *Proceedings of the First International Workshop on Software Engineering Education Based on Real-World Experiences*, pages 29–32. IEEE Press, 2012.

[28] Amir Mujkanovic and Andreas Bollin. Improving learning outcomes through systematic group reformation-the role of skills and personality in software engineering education. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2016 IEEE/ACM*, pages 97–103. IEEE, 2016.

[29] Todd Sedano, Arthi Rengasamy, and Cécile Péraire. Green-lighting proposals for software engineering team-based project courses. In *Software Engineering Education and Training (CSEET), 2016 IEEE 29th International Conference on*, pages 175–183. IEEE, 2016.

[30] Guillermo Rodríguez, Álvaro Soria, and Marcelo Campo. Measuring the impact of agile coaching on students' performance. *IEEE Transactions on Education*, 59(3):202–209, aug 2016.

[31] Hiroshi Igaki, Naoki Fukuyasu, Sachio Saiki, Shinsuke Matsumoto, and Shinji Kusumoto. Quantitative assessment with using ticket driven development for teaching scrum framework. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*. ACM Press, 2014.

[32] David Delgado, Alejandro Velasco, Jairo Aponte, and Andrian Marcus. Evolving a project-based software engineering course: A case study. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, nov 2017.

[33] Pasquale Salza, Paolo Musmarra, and Filomena Ferrucci. Agile methodologies in education: A review. In *Agile and Lean Concepts for Teaching and Learning*, pages 25–45. Springer, 2019.

[34] Willy Wijnands and Alisa Stolze. Transforming education with eduscrum. In *Agile and Lean Concepts for Teaching and Learning*, pages 95–114. Springer, 2019.

[35] Sergio Donizetti Zorzo, Leandro de Ponte, and Daniel Lucredio. Using scrum to teach software engineering: A case study. In *2013 IEEE Frontiers in Education Conference (FIE)*. IEEE, oct 2013.

[36] J.-G. Schneider and Rajesh Vasa. Agile practices in software development - experiences from student projects. In *Australian Software Engineering Conference (ASWEC06)*. IEEE, 2006.

[37] James B. Fenwick. Adapting XP to an academic environment by phasing-in practices. In *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, pages 162–171. Springer Berlin Heidelberg, 2003.

[38] Joe Bergin, James Caristi, Yael Dubinsky, Orit Hazzan, and Laurie Williams. Teaching software development methods: the case of extreme programming. In *ACM SIGCSE Bulletin*, volume 36, pages 448–449. ACM, 2004.

[39] Zainab Masood, Rashina Hoda, and Kelly Blincoe. Adapting agile practices in university contexts. *Journal of Systems and Software*, 144:501–510, oct 2018.

[40] Matthias M Müller and Walter F Tichy. Case study: extreme programming in a university environment. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 537–544. IEEE Computer Society, IEEE Comput. Soc, 2001.

[41] David F Rico and Hasan H Sayani. Use of agile methods in software engineering education. In *Agile Conference, 2009. AGILE'09.*, pages 174–179. IEEE, 2009.

[42] Orni Meerbaum-Salant and Orit Hazzan. An agile constructionist mentoring methodology for software projects in the high school. *ACM Transactions on Computing Education*, 9(4):n4, 2010.

[43] Lukas Alperowitz, Dora Dzvonyar, and Bernd Bruegge. Metrics in agile project courses. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 323–326. IEEE, 2016.

[44] Andreas Meier, Martin Kropp, and Gerald Perellano. Experience report of teaching agile collaboration and values: Agile software development in large student teams. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. IEEE, apr 2016.

[45] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[46] Jonathan E Cook and Alexander L Wolf. Software process validation: quantitatively measuring the correspondence of a process to a model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(2):147–176, 1999.

[47] Nico Zazworka, Kai Stapel, Eric Knauss, Forrest Shull, Victor R. Basili, and Kurt Schneider. Are developers complying with the process. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM 10*. ACM Press, 2010.

[48] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[49] James Taylor. Study on the best uses of technology in support of project-based learning. *arXiv preprint arXiv:1712.06034*, 2017.

[50] Alexey Zagalsky, Joseph Feliciano, Margaret-Anne Storey, Yiyun Zhao, and Weiliang Wang. The emergence of github as a collaborative platform for education. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 1906–1917. ACM, 2015.

[51] Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.

[52] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218, 2016.

[53] Philip M Johnson and Hongbing Kou. Automated recognition of test-driven development with Zorro. In *Agile Conference (AGILE), 2007*, pages 15–25. IEEE, 2007.

[54] Jennifer Campbell, Stan Kurkovsky, Chun Wai Liew, and Anya Tafliovich. Scrum and agile methods in software engineering courses. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE 16*. ACM Press, 2016.

[55] Nils Brede Moe, Torgeir Dingsøyr, and Tore Dybå. A teamwork model for understanding an agile team: A case study of a scrum project. *Information and Software Technology*, 52(5):480–491, may 2010.

[56] Yngve Lindsjørn, Dag I.K. Sjøberg, Torgeir Dingsøyr, Gunnar R. Bergersen, and Tore Dybå. Teamwork quality and project success in software development: A survey of agile development teams. *Journal of Systems and Software*, 122:274–286, dec 2016.

[57] Dana E. Sims, Eduardo Salas, and C. Shawn Burke. Is there a “big five” in teamwork?, 2004.

[58] Terry L Dickinson and Robert M McIntyre. A conceptual framework for interprofessional teamwork. In *Interprofessional Teamwork for Health and Social Care*, chapter 4, pages 57–76. John Wiley & Sons, Ltd, 2010.

[59] Steve W. J. Kozlowski. Enhancing the effectiveness of work groups and teams: A reflection. *Perspectives on Psychological Science*, 13(2):205–212, dec 2017.

[60] Todd Sedano, Paul Ralph, and Cecile Peraire. Software development waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, may 2017.

[61] William Krebs. Turning the knobs: A coaching pattern for XP through agile metrics. In *Extreme Programming and Agile Methods — XP/Agile Universe 2002*, pages 60–69. Springer Berlin Heidelberg, 2002.

[62] Martin Hoegl and Hans Georg Gemuenden. Teamwork quality and the success of innovative projects: A theoretical concept and empirical evidence. *Organization Science*, 12(4):435–449, 2001.

[63] Albert Bandura, Claudio Barbaranelli, Gian Vittorio Caprara, and Concetta Pastorelli. Self-efficacy beliefs as shapers of children's aspirations and career trajectories. *Child development*, 72(1):187–206, 2001.

[64] Steven M. Elias and Scott MacDonald. Using past performance, proxy efficacy, and academic self-efficacy to predict college performance. *Journal of Applied Social Psychology*, 37(11):2518–2531, nov 2007.

[65] Christoph Matthies, Thomas Kowark, Matthias Uflacker, and Hasso Plattner. Agile metrics for a university software engineering course. In *2016 IEEE Frontiers in Education Conference (FIE)*, pages 1–5. IEEE, 2016.

[66] Rachel Cardell-Oliver. How can software metrics help novice programmers? In *Proceedings of the Thirteenth Australasian Computing Education Conference-Volume 114*, pages 55–62. Australian Computer Society, Inc., 2011.

[67] Pekka Mäkiaho, Timo Poranen, and Ari Seppi. Software metrics in students' software development projects. In *Proceedings of the 16th International Conference on Computer Systems and Technologies*, pages 75–82. ACM, 2015.

[68] Paul Ralph and Paul Kelly. The dimensions of software engineering success. In *Proceedings of the 36th International Conference on Software Engineering*, pages 24–35. ACM, 2014.

[69] Robert W Lingard. Teaching and assessing teamwork skills in engineering and computer science. *Journal of Systemics, Cybernetics and Informatics*, 18(1):34–37, 2010.

[70] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[71] E Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 564–569. IEEE, 2003.

[72] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 356–363. ACM, 2006.

[73] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[74] Christian Murphy, Swapneel Sheth, and Sydney Morton. A two-course sequence of real projects for real customers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE 17*. ACM Press, 2017.

[75] Rashina Hoda, James Noble, and Stuart Marshall. The impact of inadequate customer collaboration on self-organizing agile teams. *Information and Software Technology*, 53(5):521–534, 2011.

[76] Ryan S Baker. Stupid tutoring systems, intelligent humans. *International Journal of Artificial Intelligence in Education*, 26(2):600–614, 2016.

[77] Vanessa Echeverria, Roberto Martinez-Maldonado, Roger Granda, Katherine Chiluiza, Cristina Conati, and Simon Buckingham Shum. Driving data storytelling from learning design. In *Proceedings of the 8th International Conference on Learning Analytics and Knowledge*, pages 131–140. ACM, 2018.

[78] Albert T Corbett and John R Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction*, 4(4):253–278, 1994.

[79] Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. Deep knowledge tracing. In *Advances in neural information processing systems*, pages 505–513, 2015.

[80] Xiaolu Xiong, Siyuan Zhao, Eric G Van Inwegen, and Joseph E Beck. Going deeper with deep knowledge tracing. *International Educational Data Mining Society*, 2016.