Bridging Theory and Practice in Programming Lectures with Active Classroom Programmer

Nasser Giacaman and Giuseppe De Ruvo
Department of Electrical and Computer Engineering, The University of Auckland, New Zealand

***Abstract*—*Contribution:* Active Classroom Programmer (ACP) is a software tool that places places minimal pressure on resources, and is shown to help improve student learning while also encouraging a high degree of engagement both during and outside of programming lectures.**

***Background:* Programming is difficult for students, largely due to the myriad of ever-advancing concepts. As students gradually become stronger programmers, both within a course and within their degree, they are constantly presented with new and challenging programming concepts regardless of their expertise. While lab sessions provide an excellent opportunity for students to independently practice, this does not help them in the programming process where expert scaffolding is desired.**

***Intended outcomes:* ACP is intended to engage students with active programming exercises and develop an inductive approach to learning, focusing on developing problem-solving skills.**

***Application design:* Students need guidance in the programming strategy rather than the syntax and peculiarities of the particular programming language. ACP allows students to program alongside instructors as new concepts are introduced.**

***Findings:* Experience from two concept-rich programming courses at different levels is presented, demonstrating students engaged with ACP both inside and outside lectures to deepen their understanding of the programming concepts.**

***Index Terms*—Active learning, educational software, computer engineering, computer science, classroom, programming, engagement.**

I. Introduction

Software is becoming increasingly integrated into daily life. As the industry demand for quality software developers continues to grow, students are increasingly turning to this field in hope of securing a promising career. While programming is at the heart of software development, it is not limited to computer science [1] and software [2] degrees; it is also integrated in the wider disciplines of computer engineering [3]. Historically a skill relevant to the computer scientist, it is now required for a growing number of careers. Whether it be data analysis, network design or even hardware engineering, programming is a fundamental skill that inherently includes a conceptual model for problem solving [4].

Programming is a notoriously difficult subject as it involves many abstract concepts at different levels, and students typically receive insufficient amounts of personal instruction [5]. Further disadvantaging students is that STEM disciplines still tend to follow traditional lecturing styles without incorporating active learning strategies [6]. In regards to programming, the role of the instructor should not be that of teaching the elements of programming, but rather to motivate students to be engaged [7]. This engagement is composed of two vital elements [8]:

- *Active learning:* students learn by applying older knowledge as they encounter new knowledge [9]. This is vital for long-term retention, where students make meaning of new information by relating it to existing knowledge [10].
- *Motivation:* collaboration, both with fellow students and instructors, also helps students learn [8]. *Pair programming* has long been employed in the software industry [11], as well as motivating students [12], [13].

More important than programming knowledge is the *strategy* of programming, and it has therefore been suggested that programming strategies receive more explicit attention in programming courses [14]. A common approach to target this is by exposing students to programming examples developed "live" in lectures, helping students observe the instructor's programming approach to a given problem [14]. It also nurtures an inductive rather than deductive approach to learning, by showing students that it is acceptable to make errors, and thereby better illustrating the problem-solving approach [15].

Live coding demonstrations still only focus on *what the teacher does*. Active Classroom Programmer (ACP) is (an Eclipse IDE plugin or web-based IDE) that allows allows the instructor to switch between "teaching to" and "programming *with*" students, helping students focus on the programming strategy. This process of accomplishing sub-goals (that lead to larger goals) helps promote student self-motivation, ultimately nurturing self-efficacy [16], [17]. Programming collaboratively with the instructor, a form of active learning via targeted practical exercises, helps reinforce learning outcomes. The relationship between the learning outcomes and course activities is more explicit, helping to ensure constructive alignment [18]. The contributions presented in this paper include:

- A non-intrusive approach that introduces the benefits of lab sessions in standard lectures, by bridging the gap between independent lab sessions and theoretical lectures [19].
- The ACP tool that allows the instructor full control of the lesson, minimizing student distraction and providing meaningful engagement for programming students.
- An evaluation of ACP, with valuable lessons for programming instructors looking to incorporate similar active learning exercises.
- An approach that encourages an inductive approach to learning rather than deductive, focusing on developing problem-solving skills.

II. Active Classroom Programmer

Motivation for ACP[1] grew out of an observation in programming courses, in which the instructor typically demonstrated live programming to students. Evaluations from students were largely positive, frequently commending the live coding demonstrated during lessons. It was then noticed that some students were attempting to replicate the live demonstrations on their own laptops during the lesson. While it was encouraging to see some students fully engaging in this manner, the concern was that students might be distracted replicating non-essential boilerplate code rather than tuning in to the instructor's demonstration.

As the name suggests, Active Classroom Programmer focuses on engaging programming students within the classroom environment in order to promote active learning. For it to be meaningful learning, the focus needs to be on engaging students in the same manner that would be expected from them in their own study time. In the context of programming, this essentially entails students using an Integrated Development Environment (IDE). Eclipse supports many programming languages and is actively used in industry, and it also boasts a plethora of additional plug-ins to meet endless programmer requirements. ACP has been developed as an Eclipse plug-in, but students also have the option of using the web-based ACP interface called WebIDE. This option is popular for cases where the bothers of installing development environments may be daunting for novice programmers.

A. Context of Usage

The idea behind ACP is that the instructor delivers the lesson in much the same way as usual (for example, using slides). The instructor then spontaneously releases either an unpremeditated or a pre-prepared exercise, and students are given time to complete it. The instructor then resumes the lesson in the usual manner, which may either be theoretical ("the next slide") or with more programming demonstrations. The important point is that the lesson always remains within the instructor's control. While students are from time to time allocated "thinking time" to work on code snippets, the instructor judges when it is time to continue the lesson. In this regards, the lesson has some elements of Peer Instruction [20]. Lectures have not suddenly turned into completely practical labs (which typically involve larger preplanned exercises spanning at least an hour, and with more teaching assistants required to help students). Here, the idea is to support a non-intrusive tool that does not require any dramatic shift in the instructor's preferred teaching approach.

In-Lecture Learning: The nature of the code snippets offered for students to complete is at the instructor's discretion. Maybe the exercise includes incomplete or incorrect programming statements. Rather than merely asking students "what's wrong with this?" and waiting for (often the same) students to answer, the instructor instead right-clicks on the project and gives it a short descriptive tag (for example, "fix the compilation error"). Students then "sync" to the same tag on their corresponding project. Within seconds of the instructor uploading the latest project snapshot, students immediately have the exact content as the instructor. There is no need to navigate outside of the IDE, and students update to the latest sync when they wish. The instructor immediately asks students to complete the small piece of code, which typically only lasts up to a couple of minutes. During this time, the instructor should not continue presenting to the class, to ensure students can focus on the task at hand. After the specified time has passed, the instructor resumes the lesson in the usual manner by regaining attention from students. Later in the lesson, another snapshot is tagged and the process repeated. The advantage of short and quick exercises is threefold:

- Students without a laptop should not feel disadvantaged or left out, still feeling that lectures are worthwhile attending. The atmosphere remains as a "standard lecture"; students are not "in the lab" and are not expected to have a computer in front of them.
- Focus is on the instructor *guiding* students through the programming *strategy*. Since students are in a lecture, not studying independently, the aim is to capitalize on the instructor's presence and expert knowledge.
- Short and digestible exercises provide a form of sub-goals that aim to develop self-efficacy. It is important to maintain the confidence of weaker students [21], especially as the exercises generally involve freshly-presented material.

This is not to say that ACP must only be used in this manner. It is possible to use ACP with *all* students being in front of a computer (like a lab), and having larger exercises with less scaffolding from the instructor. The larger the exercises get, and the longer the "thinking time", the closer the atmosphere approaches that of an independent lab exercise. It depends on what the instructor feels is appropriate.

Limitations of Laptops in the Classroom: Instructors should be aware that encouraging students to bring laptops into the classroom might have undesirable consequences. Heightened distractions are possible as students multitask (or are in view of another student multitasking) on a laptop [22]. As laptops are contextually relevant in learning programming, misuse may be reduced by discussing technology etiquette with students and incorporating constructive uses of the laptops [22]. The instructor should keep in mind that not all students bring a laptop to class; the exercise should therefore remain visible on the projector during self-coding time, encouraging these students to use pen-and-paper or pair-program with a peer.

Alternatives to In-Lecture Live Coding: It should be acknowledged that not all teachers feel comfortable live coding in lectures and being put "on the spot" in front of students; they may want to engage their students in coding exercises, but not necessarily to develop the code spontaneously. ACP allows for an instructor to prepare code snippets in advance. When ready, the code can be released to students and merely stepped through by the instructor in class time. This not only caters for less confident instructors, but also saves time if needed.

Post-Lecture Learning: Following the completion of the lecture, all the tags and snapshots made (either during the lesson or prepared by the instructor outside of lessons) are still available for the students. This immediately provides students

[1]ACP is available for those interested to install and use at the link https://acp.foe.auckland.ac.nz

with a rich collection of practical exercises for them to review during their own time. This also has the added advantage that the instructor can reuse the same exercises for future offerings of the course, or share them with other instructors (using ACP's export and import feature). The tagged snapshots also provide a reminder of the "points of interest" in progressing the live coding demonstrations.

B. Recommended Implementation

Should an instructor be interested in implementing ACP within their courses, it is recommended that they start with a basic approach. First, the instructor should consider using ACP as a platform of ready-to-run exercises that have been prepared before the lesson. During lessons, in between slides, the instructor can switch over to ACP and reveal the code snippets to students. They can either continue to progress through the code versions and explain the code, or alternate between slides and the ACP code snippets. The next degree of incorporation would be to include code versions that are slightly incomplete. For example, this could be a program with one or two empty or buggy method stubs. There would be sufficient scaffolding in place that the instructor knows exactly what needs completing. Finally, the "extreme" degree of using ACP, is to commence exercises from empty projects. However, having a set of instructions for guidance is extremely valuable. Students should also be provided with a brief handout that explains the end goal. This works well for tutorial-style lessons in particular.

C. Limitations

Although ACP has been used in a range of programming courses, no formal evaluation has been made for its best practices. Despite this, ACP is flexible enough to be used in any manner the instructor feels most comfortable. Importantly, an instructor does not need to be confident at live coding and develop exercises from scratch in front of students. ACP can also be used as a platform to create exercises beforehand, or even import ones created by others, with the instructor merely providing students time to work on them. Future long-term studies will focus on ACP best practices, by undertaking surveys and interviews from a wider set of instructors at multiple institutions to formalize a set of evidence-based guidelines.

This future work will include not only identifying all the elements that potentially contribute to effective ACP usage, but also measuring their impact. Of particular interest is understanding how active usage in-class impacts learning compared to active usage in the student's own time. Nonetheless, student perspectives (Section III-D) suggest that students who do not use ACP in class still see its benefits for self-study. The high out-of-class usages also imply that students feel it has a positive impact, otherwise they would not use it (especially as they were not given credit for using it). Finally, although the effort required to deliver lectures using ACP has not been formally studied, it should be noted that instructors can use it in a similar manner to Peer Instruction. The minimal effort demanded is to develop ACP activities before a lesson, that they can then simply present to their students at the appropriate points in that lesson. Instructors and students both appreciate the value of runnable code examples.

III. Evaluation

ACP was used in two different types of engineering programming courses: a graduate-level compiler course with strong programming students, and a CS2 course with less experienced programmers. The evaluations include a self-report study (using anonymous questionnaires) to understand the student experience, server logs to understand engagement throughout the courses, and collective learning comparisons to understand its impact. It is difficult to summarize some of the logistical details, such as (i) how many students brought laptops to classes and participated, (ii) the length of time taken for exercises, (iii) how an instructor incorporated the exercises within the lecture, and (iv) how students participated in the post-lecture learning. While it is acknowledged that all these elements would likely contribute to the level of learning gained, these are elements that cannot easily be measured. They are also largely dependent on the instructors' preferred teaching style, and a combination of the topic being taught and how students respond to the topic. With all that said, these are all valuable elements that instructors should keep in mind when developing active learning activities in general.

A. The CS2 and Graduate-Compiler Courses

ACP has been used in a number of programming courses, but this paper details experience from the following two:

- **CS2** is an object-oriented programming and data structures course (using C++) taken by computer systems engineering, electrical and electronic engineering, mechatronics and engineering science undergraduate students in their second year of study. The course is compulsory for all these cohorts (except engineering science), and usually has over 250 students in each offering. Rather than being a programming course for majors like computer science or software engineering disciplines, this is a classical example of a programming course for non-major engineers. As such, many of the students in this course are not necessarily motivated to "be a programmer"; being less familiar with development environments, most of them preferred the ACP WebIDE interface. This also meant students could use a tablet rather than a laptop if they preferred.
- **Graduate-Compiler** is a software engineering graduate course with a mixture of fourth year undergraduate students (in their final year) and Master's-level graduate students. The course is an elective for both cohorts and does not include any formally-scheduled lab sessions. The course is programming intensive, with the expectation that students code a substantial amount to complete assignments and understand course material. Unlike the CS2 course, the students already possess a solid programming foundation and are passionate about programming. ACP was used for half the course, to teach a compiler design module using Java. Students were encouraged to

Table I
DISTRIBUTION OF MARKS FOR GRADUATE-COMPILER COURSE.

		Year 1 (without ACP)	Year 2 (with ACP)	Year 3 (with ACP)
Students enrolled		27	43	95
Dropouts/failures		5 (19%)	2 (5%)	5 (5%)
A1	Low	20%	72%	66%
	LQ	61%	89%	83%
	Med	81%	95%	90%
	UQ	95%	98%	94%
	High	98%	100%	100%
Test	Low	21%	58%	29%
	LQ	54%	76%	56%
	Med	61%	88%	69%
	UQ	71%	93%	84%
	High	86%	100%	95%
A2*	Low	10%	7% *(5%)**	8% *(6%)**
	LQ	42%	55% *(44%)**	54% *(44%)**
	Med	70%	78% *(65%)**	77% *(69%)**
	UQ	91%	99% *(84%)**	91% *(80%)**
	High	100%	100% *(100%)**	100% *(100%)**

bring their laptops to lectures with ACP installed, using Eclipse. It is worthwhile noting that ACP was specifically designed for this compiler course, as it is an inherently difficult subject to learn that requires large amounts of guided practice.

The degree to which ACP was used in the two courses differed. As the nature of the content covered in the graduate-compiler course was difficult and abstract, the lectures relied heavily on live programming and giving students plenty of opportunity to practice the concepts in almost every lesson; about 25-50% of a lesson was spent using ACP. The interaction in the CS2 course differed in that the ACP interactions were in smaller fragments to cover various fundamental programming topics. Rather than being ACP-heavy *every* lesson (unlike the graduate-compiler course), only a select few lessons in the CS2 course were ACP-heavy depending on the content.

B. Improved Learning

Learning Over Three Years (Graduate-Compiler Course): The first study looks over three offerings of the graduate-compiler course. While this inevitably means three different cohorts were involved, there is no statistically significant differences in their incoming GPA. This is especially the case since software engineering in particular is a highly competitive program at the university, and the cutoff for entry into the program is 6.5 from 9. Over the three years, Year 1 was taught without ACP while Year 2 and Year 3 were both taught with ACP. To help isolate the effects of using ACP, the following measures were taken in Year 2 and Year 3: i) the same module content was taught by the same instructor as in Year 1; ii) the same lecture slides were used, with minimal modifications made to content and format; iii) assessments were largely isomorphic, tested the same learning outcomes, with the same weightings and format and with comparable difficulty (with the exception of the final assignment, explained in this section); iv) assessments were all marked by the same assistant in each year, using a similar marking rubric.

The assessments were composed of two assignments and one test. Assignment 1 and the Test were highly isomorphic across the three years, with near-identical difficulty. However, following the grading of these two assessments in Year 2, it was decided to increase the difficulty of the final assessment (Assignment 2) to counterbalance the high marks from Assignment 1 and the Test. This final assessment expected students to perform similar tasks as in Year 1, but with extra tasks added. In this regards, Assignment 2 was less isomorphic in Years 2 and 3 compared to Year 1. Since the incoming GPAs of the cohorts over the years were not statistically significant, the decision to increase Assignment 2's difficulty was attributed to the students (in Year 2 and Year 3 with ACP) grasping the concepts better than the cohort from Year 1 (without ACP).

Table II
STATISTICAL SIGNIFICANCE OF THE COURSE ASSESSMENT DIFFERENCES OF ACP YEARS (YEAR 2 AND YEAR 3) VERSUS NON-ACP (YEAR 1), WITH THE NULL HYPOTHESIS THAT $\bar{x}_{y1} = \bar{x}_{y2}$ (OR $\bar{x}_{y1} = \bar{x}_{y3}$).

		A1	Test	A2
Year 1 (without ACP)	$\bar{x}_{y1}$	73.1	59.0	64.0
	s_{y1}	24.904	17.745	28.400
	n_{y1}	26	26	23
Year 2 (with ACP)	$\bar{x}_{y2}$	92.1	84.4	72.8
	s_{y2}	7.345	9.529	27.619
	n_{y2}	42	43	39
Year 3 (with ACP)	$\bar{x}_{y3}$	87.6	68.3	71.0
	s_{y3}	8.436	17.215	24.784
	n_{y3}	92	94	85
ANOVA		$F = 19.79$ $p < 0.0001$	$F = 24.73$ $p < 0.0001$	$F = 0.88$ $p = 0.416$
One-tailed $P(\bar{x}_{y1} < \bar{x}_{y2})$		0.0001	0.0001	0.1170
One-tailed $P(\bar{x}_{y1} < \bar{x}_{y3})$		0.0001	0.0084	0.1227

Table I displays the distribution of the achieved marks without and with ACP over the three years. The marks only include non-zero results (i.e. students not submitting are excluded). To ensure a clearer comparison, the Year 2 and Year 3 results of Assignment 2 compare marks including only components that largely overlap with the Year 1 equivalent. However, for completeness, the marks with the added tasks are also included in italicized parenthesis with an asterisk: *(parenthesis)**. While effort was made to ensure consistency for comparison over the years, for pedagogical reasons this could not be fully achieved. As a result of this increased difficulty, Assignment 2 was no longer strongly isomorphic in Year 2 and Year 3 compared to Year 1.

By incorporating ACP during lessons in Year 2 and Year 3, this provided students an opportunity to engage practically in an otherwise abstract and difficult programming subject area. To determine the statistical significance of the results in Table I, ANOVA was first applied across the three years. The results in Table II show that the differences were statistically significant for Assignment 1 and the Test. For Assignment 2, there was no statistical significance and this is attributed to the non-isomorphic changes made to Assignment 2 in order to increase its difficulty. An unpaired two sample t-test was also performed on each of the assessment components, of Year 2 (or Year 3) versus Year 1 (with ACP versus without ACP). Again, the differences are only statistically significant for Assignment 1 and the Test. Overall, the dropout/failure rate was noticeably smaller for the years with ACP (5%) compared to the year without ACP (19%).

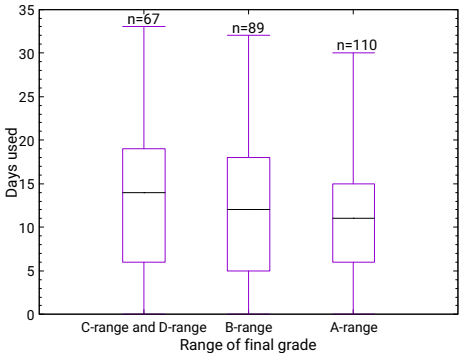


Figure 1. Range CS2-level course usage. Higher-achieving students tended to use ACP less, presumably because course content was simple enough.

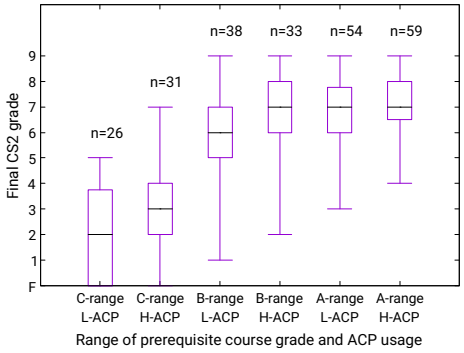


Figure 2. Within each range of incoming grades from the prerequisite course, students with higher ACP usage (H-ACP) generally performed better than peers within the same incoming ability range that used less of ACP (L-ACP).

Table III
LACK OF STATISTICAL SIGNIFICANCE OF ACP USAGE BETWEEN A-RANGE, B-RANGE AND C/D-RANGE STUDENTS WITHIN THE COURSES. THE TWO-TAILED P VALUES ARE UNPAIRED TWO SAMPLE T-TESTS FOR THE RESPECTIVE GROUPS.

	CS2	Graduate-Compiler
ANOVA	$F = 1.985$ $p = 0.1395$	$F = 0.699$ $p = 0.499$
Two-tailed $P(\bar{x}_A \neq \bar{x}_B)$	0.0954	0.4667
Two-tailed $P(\bar{x}_A \neq \bar{x}_{CD})$	0.0715	0.3143
Two-tailed $P(\bar{x}_B \neq \bar{x}_{CD})$	0.8396	0.4720

Table IV
STATISTICAL SIGNIFICANCE OF FINAL PERFORMANCE IN CS2 COURSE.

		Incoming strength (grade for prerequisite course)		
		C-range	B-range	A-range
Low ACP usage in CS2	$\bar{x}_L$	1.923	5.763	6.630
	s_L	1.719	1.684	1.629
	n_L	26	38	54
High ACP usage in CS2	$\bar{x}_H$	3.065	6.606	7.322
	s_H	1.632	1.676	1.166
	n_H	31	33	59
One-tailed $P(\bar{x}_L < \bar{x}_H)$		0.0065	0.0193	0.0051

ACP Engagement: Who Uses it More?: While the previous study looked at how ACP supported teaching an increasing depth of difficult programming concepts to students, more interesting is if ACP engagement during the semester is correlated with eventual course achievement. Fig. 1 implies that students receiving a higher grade in the CS2 course tended to use ACP less than those that received a lower grade. An ANOVA calculation on the three groups of final-grade ranges concludes that these differences are not statistically significant (F = 1.985, p = 0.1395). This is somewhat expected, as the usage of the C/D-range and B-range students in Fig. 1 are visually similar. However, when separate unpaired two sample t-tests are performed against the A-range students, a weak (yet noticeable) statistical significance is observed.

Table III summarizes the statistical significance of the usage difference between students; from this, it can only be concluded that within the CS2 course was there a hint of difference, in that stronger students tended to use ACP less. This is possibly attributable to weaker students valuing additional learning resources (which is a good thing, in that weaker students tended to opt in more for ACP). For completeness, the usage differences in the graduate-compiler course are not statistically significant. This may be due to compiler semantics being an inherently difficult topic that challenges even the academically strongest students [23]. As such, genuine immersion is vital in securing success, despite the student's academic ability.

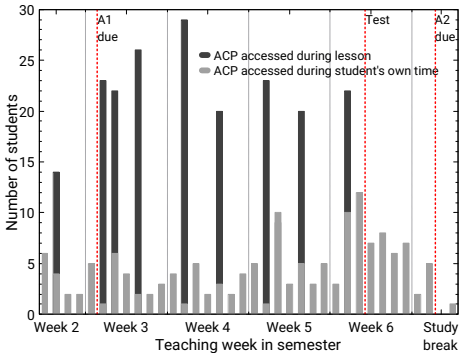
CS2 Achievement Based on Engagement: As noted above, there was a weak statistical significance suggesting that stronger students tended to use ACP less in the CS2 course. To answer the question of whether those who used ACP performed better than those that did not, the class is divided into six groups. This was based on their incoming ability (according to their performance in the prerequisite course: A-range, B-range or C-range), and whether they were low or high users of ACP (Low-ACP if used 0-11 days, High-ACP if used 12-33 days). Fig. 2 shows the final grades attained by the six groups, revealing why the usage differences of Fig. 1 (and Table III) were not statistically significant. Fig. 2 shows incoming strong/weak students (determined by the prerequisite course grade) do not always correlate to outgoing strong/weak students (as determined by the final CS2 course grade).

There are likely to be many factors contributing to success in the CS2 course, with a student's incoming ability not being the sole determinant. With that said, within each incoming ability range, those more highly engaged with ACP performed better in the course and received a higher final grade[2]. These differences are presented in Table IV, showing statistical significance in these differences. It is not possible to conclude whether this is a cause and effect observation, and ACP cannot be fully credited, as the students who were highly engaged with ACP may possibly be proactive and engaged in general with other learning material.

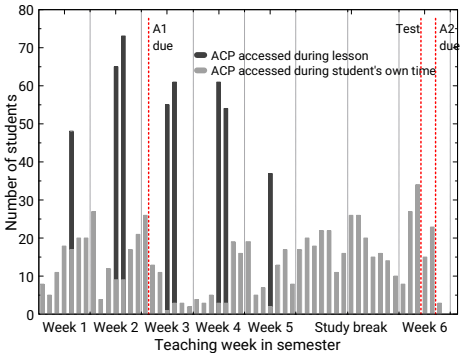
C. Access Patterns

Rather than looking at who is using ACP, this section now looks at *when* ACP is used. Namely, have students "bought" into the idea of coding *during* lessons alongside the instructor, or would they rather sit back and revise the programming exercises in their own time? Despite both courses being taught

[2]9 corresponds to A+, 8 to A, ... 2 to C, 1 to C- and 0 to D (fail).



(a) Graduate-compiler course, Year 2 (43 students enrolled)



(b) Graduate-compiler course, Year 3 (95 students enrolled)

Figure 3. Daily number of unique students accessing ACP during the two offerings when ACP was incorporated in the graduate-compiler course. Overall, 35% of the accesses in Year 3 were outside of lecture times (54% if excluding the non-teaching weeks). This suggests that students strongly valued the utilization of ACP *during lessons*, although there was also still plenty of self-study usage.

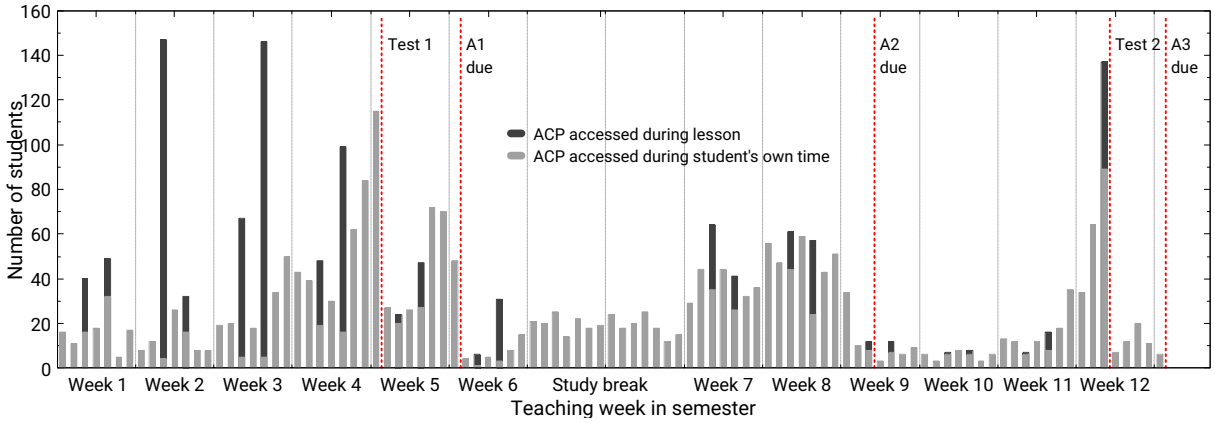


Figure 4. Daily number of unique students accessing ACP during the CS2 course (266 students enrolled). Overall, only 24% of the accesses were during class times (27% if excluding the non-teaching weeks). Considering that over 70% of ACP interaction was outside of class time, this seems to suggest that students felt there were benefits in using ACP *during self-study*. The notable decrease in usage during lessons is attributed to students opting to use ACP at their own pace, and preferred concentrating on the instructor creating the ACP exercises. The high usage prior to assessments is also another indicator that students value ACP for revision purposes. The notable low usage in weeks 9 and 10 were due to no *new* material being delivered through ACP.

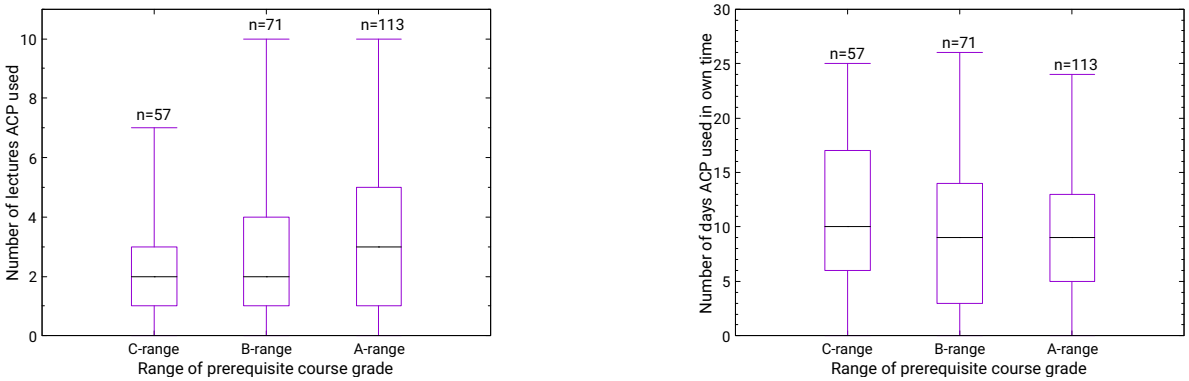
by the same instructor, ACP seems to have been embraced differently in each course. Fig. 3(a) and Fig. 3(b) illustrate the daily access patterns throughout the graduate-compiler course in Year 2 and Year 3 respectively. During the teaching weeks, over half the accesses were during the lessons. This is attributed to a number of factors, including (i) high lecture attendance rates[3], (ii) a strong cohort highly engaged in general, and (iii) students having accepted that absorbing the difficult content requires active participation during lessons.

In the case of the CS2 course, Fig. 4 illustrates how students generally felt less inspired to engage actively during lessons; over the teaching weeks, barely a quarter of the accesses were during lessons. Despite a solid participation during earlier lessons (e.g., weeks 2 and 3), the eventual attitude in this course was that students were more content observing the instructor progress through the ACP activities. This was especially noticed in weeks 7 and 8, where students opted for using ACP in self-study time. Compared to the graduate-compiler course, the CS2 typically sees lower lecture attendance rates[4] and students tend to feel less inspired towards programming.

Fig. 5(a) breaks down the *in-lecture* engagement based on the incoming grades of the prerequisite course for CS2, while Fig. 5(b) shows the equivalent breakdown for the self-study engagement. The in-class difference between the students is not considered to be statistically significant, with an unpaired two sample t-test revealing a one-tailed $P(\bar{x}_C < \bar{x}_A)$ = 0.1803 that the A-range students are more engaged than the C-range students during lessons. Although this is not statistically significant, some weaker students have noted that coding exercises were too challenging to keep up with during the lectures, and it is possible weaker students attended lectures less often than stronger students [24], [25]. What is slightly more certain, is that the weaker students were noticeably more active with ACP as a self-study tool outside lecture times; an unpaired two sample t-test reveals a one-tailed $P(\bar{x}_A < \bar{x}_C)$ = 0.04835. This may be because they prefer progressing the exercises at their own pace [26], or simply because stronger

[3]For the graduate-compiler course, an attendance rate of 70% to 80% is typical, and considered good with respect to university averages.

[4]For the CS2 course, an attendance rate of 40% to 50% is unfortunately typical, and considered standard with respect to university averages.



(a) In-lecture usage (b) Own time usage

Figure 5. Based on incoming grades from the prerequisite course, (a) in-lecture engagement versus (b) own-time engagement for the CS2 course.

students felt content with the material and did not feel the necessity to (further) practice in their own time [27].

D. Student Perspectives

1) Self-Reporting from Graduate-Compiler Students: At the conclusion of the graduate-compiler course, students were invited to complete an anonymous in-depth survey dedicated to their experience using ACP. Of the 95 students enrolled (during Year 3), 91 students completed this anonymous questionnaire. Table V categorizes the responses of the multi-choice questions, to help understand how students felt a practical lecture structure best supported their learning. Students overwhelmingly felt that using ACP contributed to better understanding of the course content, especially improving their confidence (Q1 to Q3). The high level of engagement students experienced (Q4, Q5, Q17) was genuine willingness (Q6, Q7). Students generally did not feel using ACP in the classroom was a distraction (Q8), and some even felt it improved concentration (Q9).

Another important aspect was understanding *how* the ACP workflow should be created. Students were strongly appreciative of having "quiet time", which means the instructor needs to stop talking to allow students to concentrate on the exercises (Q10). Ensuring smooth integration, without requiring separate external tools, was also important (Q11). This would be even more important for programming courses where students would not be familiar with code versioning systems. Only 20% of the students felt that the amount of time dedicated to practical ACP coding should be one-fifth or less of the lecture time (Q19). About 45% of the students wanted half the lecture time dedicated to practical coding exercises, while a further 35% of students felt that only one third of the lecture would be sufficient.

The importance of instructor guidance is raised in Q21, as students generally prefer medium-sized problems that allow scaffolding to the programming strategy; in this regards, a large lab-style exercise is not an appropriate substitute for lectures. The potential of ACP to help students during their own study time, as suggested in the previous usage patterns, is also confirmed (Q12, Q13). The fundamental feature of ACP, to retain the various snapshots of a project as it is developed, was highly rated (Q15). Also encouraging, is that most students did not feel inconvenienced bringing their laptop to lessons (Q16). Some open-ended questions were also asked:

- *"What was the best part of using the ACP tool?"* Most students referred to ACP's versioning system, and also valued the opportunity to practice with relevant exercises that are ready-to-run. Other selected comments:
  - *"Better engagement in learning, less prone to get distracted and zone out."*
  - *"Instantly getting the code from the screen to my PC. Saved me from typing EVERYTHING. I could pay attention, not having to worry about how my code differs from the instructor for the next activity."*
  - *"Understanding the difficult concepts of the course was much easier when using ACP."*
- *"In what way, if any, do you feel that the tool assisted your learning?"* Most responses for this question commented how ACP assisted preparation for assessments, and having practical examples available for an abstract subject was also helpful. Other comments include:
  - *"I always learn so much more by being practical and doing it myself, so ACP was great for that."*
  - *"Best way to learn software is by practice. Lecture time is optimal."*
  - "*I didn't have to study much for the test due to full participation in class. I grasped onto concepts fast due to practical exercises*."

  The following comment illustrates that, even though some students opted not to use ACP during lectures, they still valued it for their own study time:
  - *"Seeing the program in lecture does not help to understand it clearly, hence, I always use ACP after class to try and understand the class better."*

2) Self-Reporting from CS2 Students: A formative evaluation was released to the CS2 students mid-semester, delivered as an anonymous online survey. Only 71 of the students responded (roughly 30% response rate), but it still provided insight confirming some of the observations from the logs. Overall, 51% and 34% of the students strongly agreed and agreed respectively that "*The ACP activities help to develop my understanding of the subject*". When asked the open

Table V
MULTI-CHOICE RESPONSES FROM 91 STUDENTS IN THE ANONYMOUS ACP QUESTIONNAIRE (COMPILER GRADUATE COURSE).

	SA	A	N	D	SD
Developing self-efficacy of subject matter					
Q1. Using the ACP tool helped deepen my understanding of the subject	57%	39%	3%	1%	0%
Q2. Using the ACP tool, I was able to grasp the course material presented in class a lot faster	44%	39%	15%	2%	0%
Q3. I was more confident about the course material after using ACP	47%	42%	9%	2%	0%
Engagement and concentration during programming lectures					
Q4. ACP improved my engagement in the classroom	40%	39%	16%	5%	0
Q5. Assuming I had a laptop with ACP on it, I would try to use it for every opportunity that was presented by the instructor	47%	49%	3%	1%	0%
Q6. Although encouraged to use ACP, I did not feel forced to use it. I used it because I wanted to and because it helped me	47%	40%	12%	0%	1%
Q7. I wish this tool would be used in other programming courses	56%	33%	9%	1%	1%
Q8. I found using the tool was a distraction in the classroom environment, please stop using it	1%	2%	8%	37%	52%
Q9. ACP helped me maintain my concentration, by anticipating the next exercise	20%	34%	35%	9%	2%
Workflow					
Q10. Having "quiet time" for an exercise is important (i.e. the instructor should not attempt to continue explaining during this time)	34%	38%	16%	11%	1%
Q11. Using something like SVN/Git would make more sense instead of an Eclipse plugin	6%	11%	39%	29%	15%
Scaffolding for self study					
Q12. The tool helped me prepare for assessments	55%	35%	4%	6%	0%
Q13. I find it helpful to use ACP outside of the classroom in my own time	44%	35%	13%	6%	2%
Usability					
Q14. The tool was simple, easy to use and non-intrusive	44%	52%	3%	1%	0%
Q15. I found the versioning system a useful feature of the ACP tool	64%	22%	12%	2%	0%
Q16. I found it rather a nuisance and inconvenience to bring my laptop to university just to use ACP	2%	18%	25%	30%	25%

	(i)	(ii)	(iii)	(iv)	(v)
Q17. For this course, during the lectures you used the ACP tool: *(i) Every lecture and exercise (ii) Every lecture, but not every exercise (iii) Most lectures (iv) Very rarely (v) Never*	33%	26%	37%	4%	0
Q18. Whenever you decided NOT to use ACP in lessons, this was because: *(i) No laptop (ii) Laptop technical difficulties (iii) Exercises too difficult to attempt (iv) Exercises too easy (v) Not applicable*	6%	16%	13%	5%	60%
Q19. In a typical lesson, how much of it would you like to see dedicated for practical ACP coding? *(i) None, all theory (ii) About 10% of class time (iii) About 20% (iv) About 30% (v) About 50%*	0%	2%	18%	35%	45%
Q20. Assuming a fixed total amount of time dedicated to coding in class, how should exercises to be spread throughout the lesson? *(i) x20 small, quick and easy (ii) x10 medium (iii) x5 medium to large (iv) x2 large (v) Mixture of differing difficulties*	6%	19%	28%	18%	29%
Q21. How big should each of the in-class practical exercises be? *(i) 10-30 seconds (ii) 30-60 seconds (iii) 1-2 minutes (iv) 2-3 minutes (v) 3-5 minutes*	6%	8%	28%	24%	34%
Q22. Regarding the balance in the classroom between the theoretical (i.e. just using powerpoint slides) and practical (using ACP for in-class coding exercises), how should the delivery of the course material be modified? *(i) Significantly less theory, more practical (ii) Slightly less theory (iii) Just right (iv) Slightly less practical (v) Significantly less practical*	2%	11%	67%	18%	2%

question "*What aspects of the course are most helpful for your learning?*", over 70% of the comments credited ACP. When asked "*What changes to the course would you most like to see?*", some commented that the ACP activities could be slowed down. This strengthens the assumption that weaker students may struggle to keep up with the coding in-class, and hence why they opt to practice at their own pace afterward. This is further evidenced in the open-ended question section, with many comments confirming that it is helpful watching the lecturer during lessons, but students prefer to code in their own time as keeping up during lessons was sometimes difficult:

> *"I am sometimes unable to complete the tasks fast enough in lectures. They are very useful to go through in my own time for concept consolidation."*

In addition to the above mid-semester formative evaluation, a summative course evaluation was also released to the CS2 students at the semester conclusion. Being the official course evaluation, 165 of the 266 enrolled students responded. As a course evaluation, the focus was a holistic evaluation of the course itself rather specifically about ACP. Despite this, students were overwhelmingly in support of ACP, despite not being prompted about it. Of the optional open-ended comments for *"What was most helpful for your learning?"*, 58 of the 110 comments explicitly credited ACP. Of the optional open-ended comments for *"What improvement(s) would you like to see?"*, eight of the 103 comments explicitly stated they wanted more ACP exercises in lectures. Only seven of the 103 improvement suggestions were (mildly) negative about ACP, in that they requested that the ACP exercises be simpler in nature or that the in-class coding pace be slowed down.

IV. RELATED WORK

ACP can be placed in the broader literature of active learning [28]. Many researchers have explored different ways of delivering lectures in order to improve student engagement and comprehension of difficult concepts. In such efforts, some form of active learning is promoted over passive learning. However, despite the notable increased student satisfaction of active learning over passive learning, it is difficult to quantify improved cognitive outcomes [29]. Regardless of this challenge, active learning is an inherent part of computer science education, with a wide range of activities that engage students. Cooperative learning is a subset of active learning, where students work in groups, with particular emphasis being placed on the importance of cooperation among the small groups [30]. There are many factors that influence how computer science courses are taught, and how instructors decide to achieve the desired learning outcomes [1]. It is generally the responsibility

of the instructor to present the material at an appropriate pace for most students, and it is inevitably a challenge catering for a wide range of student abilities [31].

Apart from just adding labs to standard lectures [32], computer science education considers another two ways to incorporate labs: lab-based teaching [33] and hybrid labs [34]. Walker [35] uses a lab-based approach combining and integrating classroom and lab activities. With this format, lectures are rare and students work together most of the time. Similar to lab-based teaching is flipped learning, where typical "homework tasks" are undertaken during classes with the instructor's guidance while students watch video lectures outside of class time [36]. One of the challenges of flipping classes, or lab-based teaching, is the non-trivial changes to teaching style. The instructor must alter their way of teaching, carefully preparing new material for each of the sessions in order to reach the same goals of a standard lecture [37]. This is a rather big shift from the traditional teaching format, much like studio-based learning [38], [39], where lectures are turned into labs and students expected to use their own laptops (or one is provided).

An evolution of added labs is the one proposed by Urness [34] where students are provided with a lab exercise a few days before the lesson and can submit their solutions beforehand (similar to an open lab [40]), or they can attend the lesson where an instructor is available to help (similar to a closed lab [41]). Since students can submit solutions without attending, hybrid labs do not require many instructors or teaching assistants during lab sessions. If hybrid labs can be seen as a diverse teaching style technique, ACP is a framework to create a bridge between theoretical lectures and full independent lab sessions. It allows for finer intertwining of "theory" and "practice" during lectures, where the instructor-led approach allows changing the pace or style of the lecture "on the fly" and as appropriate for the student cohort.

Working examples are a big help for students to understand difficult concepts [42]. Not only does ACP provide a platform for (optional) in-class engagement, but the instructor can directly align exercises with the course objectives. Students end up with editable and highly relevant examples to practice and execute again. If the lesson requires focusing more on theory or practice, the instructor is able to alternate during the lesson rather than committing beforehand. In traditional lab sessions, students are always "the driver", never playing "the observer" role which is equally a valuable pair-programming role. In such cases, this independent practice has limited opportunity for students to be in-sync with the instructor and be guided by a strategic programming approach [14].

Process oriented guided inquiry learning (POGIL) is another student-centered teaching approach and may be supported by ACP when using programming exercises. The overarching idea of POGIL is to support students in working in groups on inquiry-based activities and guide them to discover concepts [43]. These activities, aligned with the learning cycle, may add additional skills such as teamwork and written/oral production. Similar to ACP is the Integrated Student-Lecturer Engagement Design Framework [44], which includes various teaching techniques. InClass Assistant is a tool that supports class participation by increasing the in-class communication between students and instructor, by providing feedback on class activities [45]. Parson's Programming Puzzles are also known to be effective in helping first-year students develop fundamental code writing skills [46], but may be difficult to extend to more advanced programming courses.

The aspiration behind ACP is very similar to that of Peer Instruction [20] (PI), moving away from the traditional model where class time is used for information transfer and encouraging students to make sense of concepts *during* class time [47]. PI has been successfully used in computer science courses [48], encouraging students to understand how to apply concepts rather than merely replicate the instructor's examples. While ACP varies in that it does not focus on peer discussions, it promotes core aspects of PI such as incorporating mini-lectures, posing exercises to students, providing think time, and discussing the final solution aided by the instructor.

Some online platforms allow students to code exercises in the convenience of a web browser, such as Python Tutor [49] and CodingBat [50]. A web browser may not be ideal (or even feasible) for non-trivial projects that demand non-standard libraries, a GUI interface or custom IDE plugins; ACP handles this, as it also offers an Eclipse IDE plug-in. ACP is unique in that it allows students to sync rapidly with the instructor as the exercises are developed. Finally, integrated versioning systems may provide students with a useful tool to follow the instructor's steps during the demonstration. While using an actual version control tool like Git [51] may be considered, it must be remembered that the exercises are to be undertaken in the context of a time-limited lesson, where downtime should be minimized. Especially in the situation of teaching novice programmers [52], the added step to sync versions is unnecessarily complicated and a distraction from the main learning objective of the exercise.

V. Conclusions

With the increased ubiquity of software applications, programming is becoming an ever more relevant subject area. As programming involves a wide range of abstract concepts, students become disengaged in the course due to the difficulty. Programming is one of those skills that is best learnt by practice, typically performed in the lab; but in order to practice, students need to learn the fundamentals, typically taught in lectures. This separation of lectures and labs is large and weakens the constructive alignment between the learning outcomes and the associated activities. ACP bridges this gap, by allowing instructors to introduce the benefits of practical labs while also maintaining the benefit of lectures. Evaluations show strong support for using ACP to help students, with minimal changes to an instructor's preferred lecturing approach. This study sheds light on experiences teaching CS2 programming for engineering non-majors, as well as a complex graduate-level compiler course for software engineering students.

References

[1] ACM and IEEE, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*, 2013.

[2] ACM and IEEE, *Software Engineering Curricula 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, 2014.

[3] ACM and IEEE, *Computer Engineering Curricula 2016: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering*, 2016.

[4] J. M. Wing, "Computational thinking," *Commun. ACM*, vol. 49, pp. 33–35, Mar. 2006.

[5] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," *SIGCSE Bull.*, vol. 37, pp. 14–18, June 2005.

[6] S. Freeman, S. L. Eddy, M. McDonough, M. K. Smith, N. Okoroafor, H. Jordt, and M. P. Wenderoth, "Active learning increases student performance in science, engineering, and mathematics," *Proceedings of the National Academy of Sciences*, vol. 111, no. 23, 2014.

[7] T. Jenkins, "Teaching programming – a journey from teacher to motivator," in *The 2nd Annual Conference of the LSTN Center for Information and Computer Science*, 2001.

[8] J. Biggs and C. Tang, *Teaching for quality learning at university*. Open university press, 3rd ed., 2007.

[9] R. E. Mayer, "The psychology of how novices learn computer programming," *ACM Comput. Surv.*, vol. 13, pp. 121–141, Mar. 1981.

[10] E. F. Barkley, *Student Engagement Techniques: A Handbook for College Faculty [Paperback*. Jossey-Bass, 2009.

[11] L. Williams, R. Jeffries, R. R. Kessler, and W. Cunningham, "Strengthening the case for pair programming," *IEEE software*, vol. 17, no. 4, pp. 19–25, 2000.

[12] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik, "Improving the CS1 experience with pair programming," *SIGCSE Bull.*, vol. 35, pp. 359–362, Jan. 2003.

[13] L. Williams, E. Wiebe, K. Yang, M. Ferzli, and C. Miller, "In support of pair programming in the introductory computer science course," *Computer Science Education*, vol. 12, no. 3, pp. 197–212, 2002.

[14] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[15] R. Felder and L. Silverman, "Learning and teaching styles in engineering education," *Engineering education*, vol. 78, no. 7, pp. 674–681, 1988.

[16] A. Bandura and D. H. Schunk, "Cultivating competence, self-efficacy, and intrinsic interest through proximal self-motivation," *Journal of personality and social psychology*, vol. 41, no. 3, pp. 586–598, 1981.

[17] B. J. Zimmerman, "Self-efficacy: An essential motive to learn," *Contemporary educational psychology*, vol. 25, no. 1, pp. 82–91, 2000.

[18] J. Biggs, "Enhancing teaching through constructive alignment," *Higher Education*, vol. 32, no. 3, pp. 347–364, 1996.

[19] M. A. Brito and F. de Sá-Soares, "Assessment frequency in introductory computer programming disciplines," *Computers in Human Behavior*, vol. 30, pp. 623–628, 2014.

[20] E. Mazur, "Peer instruction: getting students to think in class," in *AIP Conference Proceedings*, vol. 399, pp. 981–988, 1997.

[21] A. K. Lui, R. Kwan, M. Poon, and Y. H. Cheung, "Saving weak programming students: applying constructivism in a first programming course," *ACM SIGCSE Bulletin*, vol. 36, no. 2, pp. 72–76, 2004.

[22] F. Sana, T. Weston, and N. J. Cepeda, "Laptop multitasking hinders classroom learning for both users and nearby peers," *Computers & Education*, vol. 62, pp. 24–31, 2013.

[23] S. R. Vegdahl, "Using visualization tools to teach compiler design," vol. 16, no. 2, pp. 72–83, 2000.

[24] C. Walbeek, "Does lecture attendance matter? some observations from a first-year economics course at the university of cape town," *South African Journal of Economics*, vol. 72, no. 4, pp. 861–883, 2004.

[25] S. R. Tiwari and L. Nafees, *Innovative Management Education Pedagogies for Preparing NextGeneration Leaders*. IGI Global, 2015.

[26] S. Ritter, M. Yudelson, S. E. Fancsali, and S. R. Berman, "How mastery learning works at scale," in *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*, pp. 71–79, ACM, 2016.

[27] A. Cook-Sather, B. Clarke, D. Condon, K. Cushman, H. Demetriou, and L. Easton, *Learning from the student's perspective: A sourcebook for effective teaching*. Routledge, 2015.

[28] J. J. McConnell, "Active and cooperative learning: tips and tricks (part i)," *ACM SIGCSE Bulletin*, vol. 37, no. 2, pp. 27–30, 2005.

[29] N. Michel, J. J. Cater, and O. Varela, "Active versus passive teaching styles: An empirical study of student learning outcomes," *Human Resource Development Quarterly*, vol. 20, no. 4, pp. 397–418, 2009.

[30] J. J. McConnell, "Active and cooperative learning: further tips and tricks (part 3)," *SIGCSE Bulletin*, vol. 38, no. 2, pp. 24–28, 2006.

[31] M. I. Reid, L. R. Clunies-Ross, B. Goacher, and C. Vile, "Mixed ability teaching: Problems and possibilities," *Educational Research*, vol. 24, no. 1, pp. 3–10, 1981.

[32] D. Cowden, A. O'Neill, E. Opavsky, D. Ustek, and H. M. Walker, "A c-based introductory course using robots," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, (New York, NY, USA), pp. 27–32, ACM, 2012.

[33] V. Dolgopolovas, L. Savulionienė, and V. Dagienė, "Enhancing students' motivation in the inverted CS2 course: A case study," in *Proceedings of the International Conference on e-Learning*, pp. 137–141, 2014.

[34] T. Urness, "A hybrid open/closed lab for cs 1," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 46–51, ACM, 2017.

[35] H. M. Walker, "Lab-based courses with the 3 C's: content, collaboration, and communication," *Inroads*, vol. 8, no. 4, pp. 26–29, 2017.

[36] C. Herreid and N. Schiller, "Case studies and the flipped classroom," *Journal of College Science Teaching*, vol. 42, no. 5, pp. 62–66, 2013.

[37] H. M. Walker, "A lab-based approach for introductory computing that emphasizes collaboration," in *Computer Science Education Research Conference*, pp. 21–31, Open Universiteit, Heerlen, 2011.

[38] M. Barak, J. Harward, and S. Lerman, "Studio-based learning via wireless notebooks: a case of a Java programming course," *International Journal of Mobile Learning and Organisation*, vol. 1, no. 1, 2007.

[39] D. Hendrix, L. Myneni, H. Narayanan, and M. Ross, "Implementing studio-based learning in CS2," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, pp. 505–509, 2010.

[40] M. Thweatt, "Csi closed lab vs. open lab experiment," *ACM SIGCSE Bulletin*, vol. 26, no. 1, pp. 80–82, 1994.

[41] R. P. Mihail and K. Roy, "Closed labs in programming courses: A review," in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering*, 2016.

[42] M. Guzdial, "How we teach introductory computer science is wrong," *Blog at Communications of the ACM. Trusted insights for Computing's Leading Professionals*, 2009.

[43] H. Hu and T. Shepherd, "Using POGIL to help students learn to program," *Transactions on Computing Education*, vol. 13, no. 3, 2013.

[44] Y. C. Huei, "Student engagement and learning using an integrated student-lecturer engagement design framework," in *International Conference on Teaching, Assessment and Learning*, pp. 79–85, 2014.

[45] Y. Martínez-Treviño, "InClass assistant, enhancing student class participation," in *Frontiers in Education Conference*, 2016.

[46] D. Parsons and P. Haden, "Parson's Programming Puzzles: a fun and effective learning tool for first programming courses," in *Proceedings of the 8th Australasian Conference on Computing Education*, 2006.

[47] C. H. Crouch and E. Mazur, "Peer instruction: Ten years of experience and results," *American Journal of Physics*, vol. 69, no. 9, 2001.

[48] L. Porter, C. Lee, and B. Simon, "Halving fail rates using peer instruction: a study of four computer science courses," in *Proceeding of the technical symposium on computer science education*, 2013.

[49] P. Guo, "Python Tutor." http://pythontutor.com, 2018.

[50] N. Parlante, "CodingBat." http://codingbat.com, 2018.

[51] J. Feliciano, M.-A. Storey, and A. Zagalsky, "Student experiences using GitHub in software engineering courses," in *Proceedings of the International Conference on Software Engineering Companion*, 2016.

[52] D. Čubranić and M. A. D. Storey, "Collaboration support for novice team programming," in *Proceedings of the 2005 international ACM SIGGROUP conference on supporting group work*, pp. 136–139, 2005.

Nasser Giacaman is a Senior Lecturer in the Department of Electrical and Computer Engineering at the University of Auckland, New Zealand. His disciplinary research interest includes parallel programming, particularly focusing on high-level languages in the context of desktop and mobile applications running on multi-core systems. He also researches software engineering education by driving the development of tools and apps to help students learn difficult programming concepts.

Giuseppe "Pino" De Ruvo is a postdoctoral Research Fellow in the Department of Electrical and Computer Engineering at the University of Auckland, New Zealand. Giuseppe has a mixture of research and industry experience, while his Ph.D. interdisciplinary research applied model checking to various contexts. His research interests are in computer science/software engineering education, FLOSS, wikis, model checking, and empirical software engineering.