Evolving a Programming CS2 Course: A Decade-Long Experience Report

Nasser Giacaman
n.giacaman@auckland.ac.nz
University of Auckland
Auckland, New Zealand

Partha Roop
p.roop@auckland.ac.nz
University of Auckland
Auckland, New Zealand

Valerio Terragni
v.terragni@auckland.ac.nz
University of Auckland
Auckland, New Zealand

ABSTRACT

Despite instructors' best efforts in designing and delivering any given course, changes are likely required from time to time. This experience report presents the changes made in a second-year programming course for non-computing engineering majors over a decade's worth of effort, and the reasons behind those changes. The changes were often reactive—in response to student feedback. However, many other changes were inspired by the desire to trial new interventions in the hope of strengthening the students' positive experience. In addition to personnel and course content changes, the gradual evolvement included how labs, assignments, and activities were structured and executed. Teaching delivery evolved, along with a number of small-scale interventions that eventually became integral elements of the course. When COVID-19 demanded a sudden shift to online learning, the course was prepared to adapt quickly and successfully. The contributions here come in the form of lessons learned over the past decade: what worked, and what did not. We present the large range of changes—and their rationales—that are particularly relevant and applicable to programming courses targeting engineering students where the luxury of pedagogically-friendlier programming languages is not possible.

CCS CONCEPTS

• **Software and its engineering** → *Object oriented languages*; • **Social and professional topics** → **Computing education**; • **Applied computing** → **Education**.

KEYWORDS

Course Evolution, Course Redesign, CS2, Long-Term Reflection, Programming, Student Evaluations, Teaching Reflection.

ACM Reference Format:
Nasser Giacaman, Partha Roop, and Valerio Terragni. 2023. Evolving a Programming CS2 Course: A Decade-Long Experience Report. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569831

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada.

1 INTRODUCTION

Large programming courses[1] have always contributed additional teaching and management challenges—particularly around the grading of assessments [8]. Often large courses demand a distributed approach for components such as tutorials, lab sessions, and the grading of assignments. Distributing resources in this way will inevitably introduce inequitable inconsistencies as students might not receive the same support and treatment [15].

To further complicate matters, some engineering disciplines demand that students are taught using programming languages (such as C++) with reputations of being problematic and too complex for beginner programmers [18]. While much of the computing education community has moved to friendlier programming languages (such as Python) with the belief of its lower learning curve, this does not necessarily mean that students will struggle less due to the language alone [3]. More educational attention is needed, as such attitudes dismissing C++ are negatively impacting preparedness in key industries (such as telecommunications and embedded systems) [35].

Practical activities are highly accepted as an essential component to learning programming [32]. Well-designed assignments are attributed to contributing to the bulk of students' experience [10]. Programming assignments need to meet many criteria to be considered a success: relevance to a real-world problem, focused on the intended concepts, be challenging, interesting, and while allowing for creativity [33]. While valuable for learners [26], labs also impose their own set of challenges [5, 23].

But improving the students' overall learning journey is not limited to refining individual course components—such as assignments or lab exercises alone. Instead, a holistic approach is often required that will include many modifications to be applied over many years [36]. It is with the collective changes that general improvement may be observed, while acknowledging fluctuations in student satisfaction and learning are possible along the way. Given the long-term nature of noticeable improvements, instructor retention can play a key role [38].

In this paper, we present our accumulated experience in constantly evolving a CS2 course over almost a decade. We believe that the insights emerged from this evolution can benefit the computing education community with valuable lessons learned. Indeed, digital technologies are fast evolving and demanding constant improvements. While there are many studies that investigate innovative ways to redesign programming courses (e.g., [14, 16, 21, 36]), only few studies reflect on the continuous evolution of programming courses spanning several years [4, 29, 39]. However, such studies mainly focus on evolving course content, while this paper brings a novel reflection from an equally important aspect: lecturing style and organisation.

[1] Even 100+ can be considered large [15].

We have experienced teaching the same course for almost a decade, and witnessed the impact of those iterative changes—many of which were due to scare resources and budgets in the face of increasing student enrollments [31]. In recognition of the difficulties associated with analysing qualitative evaluations [34], we share a brief synthesis of qualitative evaluations carried out over multiple course offerings—and the resulting changes arising from that—which we believe are transferable to many programming courses. While much of the computing education literature has touched on many of the components individually, here we present substantial and continuous course redesign over almost a decade—a difficult research area that has received scant attention [20].

Our goal is to illustrate the value of long-term reflections in evolving courses in an ever-changing discipline, such as computing. We end the paper with recommendations for educators seeking a progressive and incremental evolution of their programming courses.

In summary, the paper makes the following main contributions:

- Sharing a reflection with insights and lessons learned in improving the same CS2 course over a decade.
- Provide a case study in the value of documenting accumulated changes, to help instructors avoid repeated mistakes when considering frequent modifications in their courses.
- Experiences using a pedagogically-discouraged language, such as C++, as mandated for non-computing majors.
- Sharing rationales and the resulting impact of accumulative changes spanning almost a decade.

2 RELATED WORK

Programming courses require to be continuously refined—at the very least in terms of content, to remain relevant to evolving technology [12]. Besides evolving course content, evolving lecturing style and management has proven beneficial for improving students' learning and positive experience [14]. Indeed, redesign efforts on programming courses have shown to result in higher grades and lower dropout rates [14].

Several studies investigate innovative ways to redesign programming courses (e.g., [14, 16, 21]). A particularly dominant trend has been to move away from traditional lecturing styles, and towards a more practical environment as this is recognised as key in helping students embrace difficult programming concepts [2, 13, 36]. Our experience with our CS2 course confirms this. Students responded positively to our hybrid approach of blending live coding demonstrations and traditional teaching.

This paper presents an accumulated reflection over (almost) ten years on continuously evolving a programming course. To the best of our knowledge, only a few studies present a similar reflection [4, 29, 39]. Baker and Van Der Hoek [4] report their experience designing and delivering two new software design courses. Different from our study, their reflection is limited to two years only. Ramnath and Dathan [29] and Yusof and Abdullah [39] present their experience in evolving programming courses over 10 years [29]. Both studies mainly focused on the course content, while our redesign and resulting reflection mainly involved teaching style and course management. As such, our study brings an orthogonal perspective toward programming course redesign.

3 COURSE CONTEXT

This section provides the context for the second-year programming course, including the students that take it, the teaching team, learning outcomes, content, and activities for each respective year. Table 1 presents key course components each year, which are expanded below. To assist with the context of changes discussed in Section 4, the status of course scores is also provided. The Anonymous University defines courses as "red-flagged" (👎) when they score less than 70% satisfaction rate in student evaluations; otherwise they are regarded as fine (👍); it required the teaching team three iterations before they were able to turn course ratings around. The 2020 and 2021 offerings were predominantly online due to mandatory COVID-19 lockdowns.

3.1 The Students

At the Anonymous University, there are ten specialisations in the undergraduate Bachelor of Engineering (Honours) four-year degree. This paper focuses on a compulsory second-year programming course, taught to three of these specialisations in their second year:

- Computer Systems Engineering (CSE),
- Electrical and Electronics Engineering (EEE),
- Mechatronics Engineering (ME).

Regardless of specialisation, the first year is a "general" year that involves all engineering students completing the same core courses. One such course introduces basic programming using Matlab and C (six weeks dedicated to each language), which serves as the prerequisite for this course.

3.2 Content and Learning Outcomes

C++ is acknowledged as being a complex and unforgiving programming language for novice programmers [18], yet it has always been the course's programming language of choice. Beginner friendliness is not the only factor in deciding a course's programming language [9], with industry-relevance also being an important factor [24]. C++ remains a vital language in many industries relevant to the CSE, EEE, and ME specialisations; examples include telecommunications, microelectronics, graphics, embedded systems, and aerospace [35]. This message is regularly reinforced by industry representatives as part of the institution's accreditation reviews.

The course has existed in two major phases in terms of topic content. When it was inherited in 2013, the team mostly reused existing content so as to "ease into" the course. There were initially three sections, each spanning four teaching weeks in the semester:

- OOP (Object-oriented programming), such as classes, abstraction, encapsulation, inheritance, and polymorphism.
- UML (Unified modeling language), such as use cases, class diagrams, sequence diagrams, and activity diagrams.
- DSA (Data structures and algorithms), such as abstract data types, algorithms and complexity, sets, stacks, queues, linked lists, binary trees, recursion, searching, and sorting.

At the end of 2015, it was apparent that the course was trying to do too much. Due to the large amount of content that needed to be covered, lessons were moving rapidly with the pressure of getting through the material during the allotted time. As a result, students were playing a very passive role in the lectures as new information was constantly thrown at them. To address this, 2016 saw

Table 1: Course Components Each Year

	2013	2014	2015	2016	2017	2018	2019	2020_{online}	2021_{online}
Specialisations	← CSE, EEE, ME →								ME
# Students	265	227	252	266	222	259	289	300	114
Topics	← OOP UML DSA →			← OOP DSA →					
Instructors	A B C	A B C	D B C	B	B C	E C	B C	B C	B F
Labs	10x 2%	6x 2.5%	6x 2.5%	6x 0%	6x 0%	6x 0%	6x 0%	6x 0%	6x 0%
Mini Activities	-	-	-	-	-	1x 1%	1x 1%	5x 1%	5x 1%
Assignments	20,20%	15,30%	15,15,15%	20,20,20%	5,25,10,20%	4,25,10,20%	5,24,10,20%	5,20,10,20%	5,20,10,20%
Tests	20,20%	20,20%	20,20%	20,20%	20,20%	20,20%	20,20%	20,20%	20,20%
GitHub	-	-	-	-	-	-	-	Yes	Yes
Course Score	👎	👎	👎	👍	👍	👍	👍	👍	👍

a major revamp of the course where the UML section was mostly removed, resulting in only the OOP and DSA sections remaining—each stretched to span six weeks of teaching. This change not only allowed the instructors to cover the sections at a more relaxed pace, but also it enabled the possibility of more interaction with students. Most importantly, this enabled practical hands-on opportunities of engagement to be embedded in the lectures (more in Section 4.2).

None of the instructors assigned to the course from 2013 had taught it before. Each instructor was assigned one of the sections to teach. Consequently, there were three instructors during the 2013–2015 period, while the major revamp led to involving just two instructors. The course was taught solely by B in 2016, while the revamped course was being trialled. It was then decided that only two instructors would suffice in teaching the course given the reduction of sections. Hence from 2017 onward, the course was predominantly taught by the same two instructors (B and C), with the occasional use of a replacement when one of them went on sabbatical leave.

3.3 Course Assessments

Intended as a predominantly hands-on course, most assessments involve practical programming activities. There are two invigilated time-restricted tests, always composed of multi-choice questions (MCQs). This was the case for both in-person paper-based tests (using optical mark recognition scanners), and also when online due to COVID lockdowns (using quiz software). Given the large number of students, MCQs have the benefits of quick evaluation, reliable scoring, affordability, and marking automation [7]. While MCQs have the disadvantage of dealing with lower knowledge levels, this is catered for by incorporating questions with code snippets. Prior research has demonstrated strong support for such code tracing skills being associated with writing skills [19]. Another rationale for showing code snippets (rather than "worded" MCQ questions) is that there tends to be less room for incorrect interpretation of what is being asked.

Labs (Section 4.1) involve small coding exercises, typically scoped to require 20–45 minutes to complete for most students. The exercises are aimed at reinforcing the past week's lectures, while also scaffolding students towards the larger course assignments. While scheduled lab sessions are offered for students to seek help from TAs, most students resort to online support through the course's Piazza forum (https://piazza.com)—where both peers and the teaching team can respond. The take-home assignments always demanded substantially-more programming depth, typically requiring 2–4 weeks of effort. Final submissions are run through Moss [1].

Initially the assignments were too large in scope and difficult to provide timely feedback, causing distress for many students—so they tend to now be more digestible (Section 4.3). To solve typical problems supporting assignments, Git now plays an instrumental role in the course (Section 4.5). Simple mini learning activities have also been introduced to keep students motivated and engaged (Section 4.4).

4 ISSUES AND CORRESPONDING CHANGES

This section overviews the key issues that came up over the decade, many of which resulted in changes made to the course. However, not all "issues" that came up were deemed to require changes—these are also highlighted along with the reasoning behind not making changes. Where changes were motivated by student feedback, quotes are provided to help illustrate the student voice. All quotes are from anonymous end-of-semester course evaluations.

4.1 Labs

When we first inherited the course, it followed a somewhat typical format of labs. In 2013, we used the same format as used by our predecessors: 10x weekly lab exercises, each worth 2% for a total 20% of the course. This seemed reasonable to us, particularly how we valued the importance of practical programming opportunities for students—and students also recognised this:

> *"Labs every week were the most helpful because it let me practice what I learnt." [2013]*

But despite such positive comments, an overwhelmingly larger amount of negative views surrounded labs—mostly operational:

> *"It is not very fair when people who finish early have to wait the entire lab session while others who are still doing the lab exercises get more attention from TAs. Sometimes there's not enough TAs for the lab." [2013]*

These issues have stemmed from high student enrollments as is seen in many computing courses, for both computing majors and non-majors [31]. To alleviate some of these issues, in 2014 we reduced the number of lab exercises to support more resources per lab activity (10x → 6x). It was hoped that directing more of the lab costs towards quality over quantity might help. Unfortunately, these

incremental modifications did not provide the transformation we were hoping for and the resource constraints persisted to negatively impact students' learning:

> *"I didn't feel like I could ask for help in the lab sign-offs because of all the people waiting after me, which made this course particularly difficult for me." [2015]*

Since using plagiarism-detection software provided too many false-positives (given the relative simplicity of lab exercises), brief interviews were conducted for students to be signed off for lab credits. Such environments create uncomfortable student–TA interactions when dealing with plagiarism, heightened by time constraints [30]. With all these issues in mind, 2016 saw major changes to labs. Lab activities were no longer given (direct) course credit. Instead, understanding lab exercises was assessed within the invigilated tests. This allowed the lab sessions to be transformed into *lab clinics*, where students are encouraged to attend when they would like help. TAs now played a much more supportive role: rather than be the incriminating interviewer, they were now the support system that students could reach out to. Labs now received plenty of praise:

> *"The amount of lab time we had available." [2016]*

> *"Labs—Personal conversation with tutors." [2016]*

While we recognised this was an essential move to support students, there were still the odd comments requesting compulsory labs:

> *"Consider graded labs. I know this means code getting checked—perhaps just show it works? I think it's a good idea." [2016]*

While this would encourage lab attendance and engagement, we feel the welcoming support system provided by drop-in labs—rather than feeling like an interrogation zone—makes the better impact on student wellbeing. This change has stood the test of time, where labs continue to gather feedback on both sides:

> *"The labs—the TAs were very helpful and made me understand the content a lot better." [2021]*

> *"Maybe make the labs worth something, as I found it hard to be motivated to do them." [2020]*

4.2 Teaching Delivery

Despite the issues surrounding labs, students yearned for more opportunities to practice programming:

> *"The course should be 4 hours of labs, 1 hour of lectures instead. You learn coding by doing, not falling asleep trying to watch someone else show you." [2013]*

Students also wanted more guidance on programming strategies:

> *"Spend time in lectures focusing on how to approach these more difficult labs." [2013]*

When preparing the 2014 course offering, the teaching team considered the flipped classroom approach [6] as a possible delivery model to address this issue. Despite its promises, migrating the course to use this model involved too many (well-documented) challenges [2]. The higher workload and time commitment required to make such a change was far too demanding, especially as all the instructors were relatively new to teaching the course. As a compromise, live coding demonstrations were blended with traditional teaching of the concepts. Students were also encouraged to bring a laptop to lectures, where the instructors provided them with the opportunity to try some of the exercises during the lecture time. To facilitate this, Active Classroom Programmer (ACP) [11] was trialled. This allowed the instructor to share written code directly from their IDE—and students in turn modified that shared code. Students responded positively to this, and suggested more of it:

> *"Maybe more integration with ACP throughout the whole course, it's an excellent way of learning!" [2014]*

ACP provided the perfect compromise of overcoming constraints of limited lab resources, without the challenges of a truly flipped classroom approach—and quickly become a pivotal course component:

> *"I like the live coding on ACP, and having the different versions online—this was a very good reference for learning how to apply code, and understand what each line of code does." [2020]*

4.3 Assignments

The course was always intended to be highly practical, involving substantial amounts of programming from students. As such, the course was initially designed with large assignments—roughly one to cover each half-section of the course. Initially, the main issue arising from these large assignments was that they required substantial manual effort to be graded—and there were few TAs assigned to marking. This heavily impacted the turnaround for giving students feedback:

> *"Could assignments also be returned more appropriately please? By tomorrow it will be eight weeks since we handed in Assignment 1 and still we haven't received a mark or feedback." [2013]*

To correct this, assignments were revised to be slightly smaller, with more TAs hired to help with marking. Unfortunately, assignments were actually more demanding than implied by their weights. Furthermore, a new instructor's teaching pace was too fast for most students to start off the course. Many students felt discouraged struggling with the concepts, and this was a bad way to start off the course:

> *"As for the first assignment—this was an incredibly hard assignment to actually understand. Many people lost a lot of confidence in this paper after that assignment and just gave up in this paper all together." [2015]*

As part of the 2016 major course redesign, assignments were redesigned following a *constructive alignment* approach. In addition to redesigning the assignments, this also involved redesigning the in-class activities and lessons to gear students towards the assignments:

> *"The assignments assisted in understanding and applying the concepts taught in the course very well and were very appropriate." [2016]*

More changes to assignments were made in 2017, which have persisted to their current form today. This included a much smaller 5% Assignment 1 due after only two weeks of teaching—much sooner than in the past. Collectively, and most importantly, this provides students with a massive confidence boost early on in the course. This confidence is very much needed when moving on to larger assignments:

> *" The assignments clearly supported, assessed and provided feedback on my learning which was great! Very rapid feedback on assignments and tests (returned within days of deadline) allowed me to quickly address any misconceptions." [2020]*

4.4 Mini Learning Activities

To extend the benefits initially observed with the "starting small" structure of assessments, starting in 2020 saw a further 5% of the course redirected to supporting even smaller weekly at-home activities, which we collectively termed as "*mini learning activities*". There are a total of five 1% activities, starting from the first week. Unlike other assignments and lab exercises, these mini activities were intended to not involve programming, and instead be perceived as fun and easy—yet useful. The goal is to "hook students" to the course with "tasters", and inspire them to stay focused and engaged—even if feeling somewhat programming-averse. The scope of these activities is such that they require less than an hour's effort to complete.

The first activity has tended to familiarise students with GitHub, and involves giving them a stripped-down version of Assignment 1 with the same setup—but with just very few test cases they need to pass. This mini activity achieves a lot, ensuring students:

- Have created a GitHub account, linked it to their university identity, and know how to make commits and pushes.
- Have the required programming development environment set up as needed for the semester.
- Demonstrate how to successfully submit future assignments.

Another typical mini learning activity includes students writing short reflections on their journey, with prompts for guidance:

- *Regarding learning OOP and GitHub, did it spark any new idea or create any new perspective for you?*
- *How do you describe the impact of learning OOP and GitHub on your professional future?*

The marking of the 1% activities tends to be lenient, as this not only facilitates speedy-marking, but also ensuring students engage. These activities are simple to define and can also be reused in subsequent years without concerns for plagiarism. Not to mention, they are still relevant to the course and appreciated by students:

> *"The 1% activities early on were helpful to get us familiar with the concepts in the course." [2020]*

4.5 Git

Version control was traditionally never considered as belonging to this course. Despite the growing importance of Git over the years, it still felt as if there was too much overhead in managing it for a large number of students. Creating individual repositories and adding students manually was clearly not an option. Yet, there was desire for a version control solution due to scenarios such as:

- Students submitting the incorrect files to the assignment dropbox, whether that be an old version of the assignment, or files for a completely different assignment/activity.
- Suspiciously-similar submissions according to software similarity tools, but without understanding how they evolved.
- Students allege catastrophic events resulting in last-minute data loss, such as a stolen laptop or corrupted hard drive.

While many scenarios were genuine, it was always difficult telling students "*sorry, we cannot help here without sufficient evidence*". As instructors, it was difficult making such calls—but we had a duty to the rest of the students to be fair, so could only support students with sufficient evidence. Integrating GitHub Classroom for all assignments has allowed us to support students where they had the Git history supporting their story. Students are told that we expect frequent updates to their Git repository as they progress. Logs have been useful in supporting plagiarism investigations, and dismissing unwarranted requests for an extension—such as cloning the assignment repository only the day before it is due. The most rewarding case was reinstating full marks to a student that completed their assignment on time but had their laptop stolen the day before the due date.

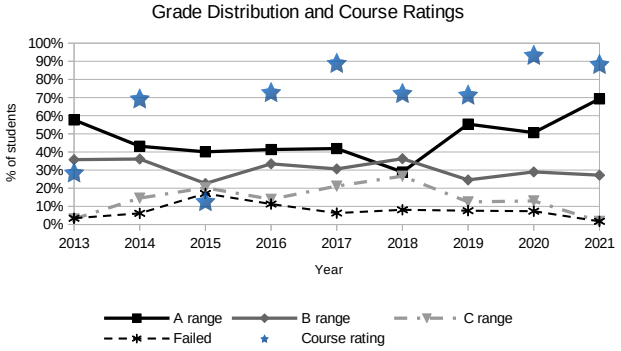

Figure 1: Distribution of grades and course rating each year.

5 REFLECTIONS

There is no such thing as the perfect course—a course will forever be evolving. What worked for us is listening to student feedback, and revising where feasible. We feel addressing the "lion's share" of student concerns each year accumulated to an overall improvement.

5.1 Course Ratings and Student Performance

Figure 1 presents the final grade distributions for each of the respective years, along with the corresponding course score resulting from the student satisfaction evaluation. As previously presented in Table 1, the course's first three years were red-flagged, while the latter years were over the 70% satisfaction rate.

Of particular interest is how students were satisfied in the two years of remote learning due to COVID-19. The are no obvious "trends", in terms of grades and course ratings. These can drop or rise as part of the natural and arbitrary environment of that particular year. What is however reassuring, is that course satisfaction is consistently better in the recent years compared to the initial years that triggered the biggest changes in the course. With that said, one needs to be aware of using surveys as a measure of course quality [17, 37].

The proportion of A-grades tend to be higher in the most recent three years. There were notable years that had different distributions of grades. In 2015, there was a lower proportion of mid-range B grades, and higher proportion of lower-range C grades—along with the most failed grades. This was the year that inspired an urgent and thorough makeover. The course was trying to teach too much content, and the first section was a fast pace for the students—resulting in struggling and loss of confidence within the first few weeks of the course. The only time there were more B grades compared to A grades was in 2018; there was nothing particularly eventful that year, and it might have just been a difference of instructors.

5.2 What Worked Well

We recognised the detrimental impact of losing student confidence early on in the course. This not only impacted the wellbeing of students trying their best, but also their ability to progress in course activities [25]. One of the best things that worked well is having smaller and easier assignments early on in the semester to hook students in. This particularly gave confidence boosts to students that enter the course nervously. The introduction of Git has also contributed to better experiences with assignments, and has rescued many students.

Based on the expectations demanded by the practical assessments, the course has evolved to provide students with the necessary support system. Rather than wasting TAs' time interviewing to sign students off, they are now dedicated to helping students overcome misconceptions. The tone of the comments in evaluations have noticeably changed, from a negative tone to a much more positive tone about both the TAs and the lab sessions as a whole. Providing students with more practical opportunities in the lecture time has also worked very well, where students are encouraged to bring their laptop to modify code snippets shared by the instructor.

5.3 What Did Not Work

Despite the lab improvements made, they are by no means perfect. Ideally we would like to see more uptake of both the lab exercises, and the assistance available in scheduled drop-in labs. When the lab clinics are under-utilised, the TAs will often use that time to respond to online questions on the course's Piazza forum. While it has been considered (and even requested by students) to give credit for completing these exercises, from our experience this only results in heightened administration effort targeted at fighting plagiarism. As a compromise, we strive to have questions in invigilated tests that stem from the lab exercises as an incentive for students to attempt them.

While the course already uses automated assessment for the grading of larger assignments, maybe it is time to use an automated tool [27], or review Git logs demonstrating incremental progress. Regardless, this will require further thinking as colluding will not be avoidable—especially in the context of a large course that cannot enforce simultaneous invigilation in computing labs, and where limited resources need to target learning.

5.4 Limitations

It is difficult to attribute course ratings to anything specific. The student evaluations are holistic on the overall student experience with the course. This is not limited to how they liked the teaching team and course assessments of a particular year, but might also include other aspects; for example, the scheduling of lectures and lab clinics (including both the rooms, dates and times)—which are out of the teaching team's control. All these factors, and plenty others, are not taken into account in this experience report. There were most definitely other changes made to the course from year to year, but here we only focus on the most problematic experiences that we faced. The impact of any individual change cannot be isolated.

5.5 Recommendations

We recommend and encourage instructors to only take on what they can, and make slow incremental steps as they can. This is in line with the literature's acknowledgement of the high efforts required [14, 36].

We have also seen benefits in embedding a few simple and low-stake 1% mini learning activities. Even with a few of them, we feel they help boost student morale without "giving too much away" (collectively only 5% in total—a small price to pay to inspire students to engage with the course concepts). Where possible, we also try and incorporate elements of gamification, as this has been shown to engage and improve student learning [22]. Usually such activities are built "in-house" by senior students, for example part of summer projects.

One of our biggest recommendations comes in the form of incorporating more in-class live programming, using whatever tools or approach the instructor feels most comfortable with. This does not need to be programming code snippets "from scratch", and can include only minor modifications to existing code snippets. Ideally students are also given opportunity to code along in the lesson. This not only provides stronger alignment to the concepts taught, but helps guide students in the programming thought process and develop their debugging skills [28]. Based on our experience, we feel this has been an effective solution when resources for lab and TAs are scarce—essentially enabling us to bring practical "labs" into the classroom.

We hope this experience report will encourage educators to perform similar long-term reflections. In particular, reflecting on the changes over a long period of time gave us a pragmatic view on what can be reasonably achieved with incremental steps. Our reflection taught us that student feedback is a powerful driving force to both motivate possible future improvements and validate changes; studying how student feedback evolved over a long time was crucial. Indeed, often negative (and positive) feedback are dictated by external factors unrelated to our best effort to improve the course. As such, we discourage using the most recent student feedback to immediately validate recent changes. Instead, observing how the students' feedback evolve over a long period of time gives a holistic view of the re-design. As such, we advocate for long-term reflections, which can provide insights that are impossible to get with a short-term reflection.

6 CONCLUSIONS

Revising courses is a natural part of teaching—particularly in computing courses as technology is changing frequently. Rather than focus on course redesign based on content, here we present our experiences changing other aspects of a second-year programming course over almost a decade. In most cases, changes were motivated by poor student satisfaction, but also often in recognition of the importance of practical programming opportunities to help students learn the concepts. The changes have been implemented iteratively over time, and sometimes required multiple revisions of the same component. Notable improvements have been shifting away from TAs "policing" students on graded lab exercises, and towards an environment where TAs are recognised for their key contribution in helping students overcome programming struggles. Teaching delivery was then adapted to provide additional practical programming exposure, while reformulating assignments aimed to target loss of confidence.

We hope that our reflective case study will inspire more instructors to document changes to their own courses, along with the rationale for those changes. We believe such reflective documentation will provide course stability in light of increasing student enrollments, teaching team turnover, and ever-changing technologies.

REFERENCES

[1] Alex Aiken. 2022. *Moss: A System for Detecting Software Similarity.* http://theory.stanford.edu/~aiken/moss

[2] Gökçe Akçayır and Murat Akçayır. 2018. The flipped classroom: A review of its advantages and challenges. *Computers & Education* 126 (2018), 334–345. https://doi.org/10.1016/j.compedu.2018.07.021

[3] Nabeel Alzahrani, Frank Vahid, Alex Edgcomb, Kevin Nguyen, and Roman Lysecky. 2018. Python Versus C++: An Analysis of Student Struggle on Small Coding Exercises in Introductory Programming Courses. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 86–91. https://doi.org/10.1145/3159450.3160586

[4] Alex Baker and André Van Der Hoek. 2009. An experience report on the design and delivery of two new software design courses. *ACM SIGCSE Bulletin* 41, 1 (2009), 519–523.

[5] João Paulo Barros, Luís Estevens, Rui Dias, Rui Pais, and Elisabete Soeiro. 2003. Using lab exams to ensure programming practice in an introductory programming course. *ACM SIGCSE Bulletin* 35, 3 (2003), 16–20.

[6] Jonathan Bergmann and Aaron Sams. 2012. *Flip your classroom: Reach every student in every class every day.* International society for technology in education.

[7] Dhawaleswar Rao CH and Sujan Kumar Saha. 2020. Automatic Multiple Choice Question Generation From Text: A Survey. *IEEE Transactions on Learning Technologies* 13, 1 (2020), 14–25. https://doi.org/10.1109/TLT.2018.2889100

[8] A. T. Chamillard and Laurence D. Merkle. 2002. Management Challenges in a Large Introductory Computer Science Course. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky) *(SIGCSE '02)*. Association for Computing Machinery, New York, NY, USA, 252–256. https://doi.org/10.1145/563340.563440

[9] Michael De Raadt, Richard Watson, and Mark Toleman. 2003. Language tug-of-war: industry demand and academic choice. In *Proceedings of the 5th Australasian Computing Education Conference (ACE 2003)*. Australian Computer Society Inc., 137–142.

[10] Todd J. Feldman and Julie D. Zelenski. 1996. The Quest for Excellence in Designing CS1/CS2 Assignments. In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education* (Philadelphia, Pennsylvania, USA) *(SIGCSE '96)*. Association for Computing Machinery, New York, NY, USA, 319–323. https://doi.org/10.1145/236452.236564

[11] Nasser Giacaman and Giuseppe De Ruvo. 2018. Bridging Theory and Practice in Programming Lectures With Active Classroom Programmer. *IEEE Transactions on Education* 61, 3 (2018), 177–186. https://doi.org/10.1109/TE.2018.2819969

[12] Richard Helps. 2007. Dancing on Quicksand Gracefully: Instructional Design for Rapidly Evolving Technology Courses. In *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education* (Destin, Florida, USA) *(SIGITE '07)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1324302.1324304

[13] Ville Isomöttönen and Ville Tirronen. 2016. Flipping and Blending—An Action Research Project on Improving a Functional Programming Course. *ACM Trans. Comput. Educ.* 17, 1, Article 1 (sep 2016), 35 pages. https://doi.org/10.1145/2934697

[14] Erkki Kaila, Einari Kurvinen, Erno Lokkila, and Mikko-Jussi Laakso. 2016. Redesigning an Object-Oriented Programming Course. *ACM Trans. Comput. Educ.* 16, 4, Article 18 (2016). https://doi.org/10.1145/2906362

[15] David G Kay. 1998. Large introductory computer science classes: strategies for effective course management. *ACM SIGCSE Bulletin* 30, 1 (1998), 131–134.

[16] Yekaterina Kharitonova, Yi Luo, and Jeho Park. 2019. Redesigning a software development course as a preparation for a capstone: An experience report. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 153–159.

[17] Antti Knutas, Timo Hynninen, and Maija Hujala. 2021. To Get Good Student Ratings should you only Teach Programming Courses? Investigation and Implications of Student Evaluations of Teaching in a Software Engineering Context. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 253–260. https://doi.org/10.1109/ICSE-SEET52601.2021.00035

[18] Michael Kölling. 1999. The problem of teaching object-oriented programming. *Journal of Object-oriented programming* 11, 8 (1999).

[19] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[20] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus). 55–106. https://doi.org/10.1145/3293881.3295779

[21] Sohail Iqbal Malik. 2018. Improvements in introductory programming course: action research insights and outcomes. *Systemic Practice and Action Research* 31, 6 (2018), 637–656.

[22] B. Marín, J. Frez, J. Cruz-Lemus, and M. Genero. 2018. An Empirical Investigation on the Benefits of Gamification in Programming Courses. *ACM Trans. Comput. Educ.* 19, 1, Article 4 (nov 2018), 22 pages. https://doi.org/10.1145/3231709

[23] Julia M. Markel and Philip J. Guo. 2021. *Inside the Mind of a CS Undergraduate TA: A Firsthand Account of Undergraduate Peer Tutoring in Computer Labs.* Association for Computing Machinery, New York, NY, USA, 502–508.

[24] Raina Mason, Tom Crick, James H Davenport, and Ellen Murphy. 2018. Language choice in introductory programming courses at Australasian and UK universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 852–857.

[25] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. 2019. A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education* 62, 2 (2019), 77–90. https://doi.org/10.1109/TE.2018.2864133

[26] Diba Mirza, Phillip T. Conrad, Christian Lloyd, Ziad Matni, and Arthur Gatin. 2019. Undergraduate Teaching Assistants in Computer Science: A Systematic Literature Review. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3291279.3339422

[27] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An Empirical Study of Iterative Improvement in Programming Assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 410–415. https://doi.org/10.1145/2676723.2677279

[28] Adalbert Gerald Soosai Raj, Jignesh M. Patel, Richard Halverson, and Erica Rosenfeld Halverson. 2018. Role of Live-Coding in Learning Introductory Programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '18)*. Association for Computing Machinery, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3279720.3279725

[29] Sarnath Ramnath and Brahma Dathan. 2008. Evolving an integrated curriculum for object-oriented analysis and design. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*. 337–341.

[30] Emma Riese, Madeleine Lorås, Martin Ukrop, and Tomáš Effenberger. 2021. Challenges Faced by Teaching Assistants in Computer Science Education Across Europe. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 547–553. https://doi.org/10.1145/3430665.3456304

[31] Linda J. Sax, Kathleen J. Lehman, and Christina Zavala. 2017. Examining the Enrollment Growth: Non-CS Majors in CS1 Courses. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 513–518. https://doi.org/10.1145/3017680.3017781

[32] Thomas Staubitz, Hauke Klement, Jan Renz, Ralf Teusner, and Christoph Meinel. 2015. Towards practical programming exercises and automated assessment in Massive Open Online Courses. In *2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. 23–30. https://doi.org/10.1109/TALE.2015.7386010

[33] Daniel E. Stevenson and Paul J. Wagner. 2006. Developing Real-World Programming Assignments for CS1. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITICSE '06)*. Association for Computing Machinery, New York, NY, USA, 158–162. https://doi.org/10.1145/1140124.1140167

[34] Carly Steyn, Clint Davies, and Adeel Sambo. 2019. Eliciting student feedback for course development: the application of a qualitative course evaluation tool among business research students. *Assessment & Evaluation in Higher Education* 44, 1 (2019), 11–24. https://doi.org/10.1080/02602938.2018.1466266

[35] Bjarne Stroustrup. 2020. Thriving in a Crowded and Changing World: C++ 2006–2020. *Proc. ACM Program. Lang.* 4, HOPL, Article 70 (2020), 168 pages. https://doi.org/10.1145/3386320

[36] Dion Timmermann, Christian Kautz, and Volker Skwarek. 2016. Evidence-based re-design of an introductory course "programming in C". In *2016 IEEE Frontiers in Education Conference (FIE)*. 1–5. https://doi.org/10.1109/FIE.2016.7757492

[37] Guannan Wang and Aimee Williamson. 2020. Course evaluation scores: valid measures for teaching effectiveness or rewards for lenient grading? *Teaching in Higher Education* 0, 0 (2020), 1–22. https://doi.org/10.1080/13562517.2020.1722992

[38] Catherine Whalen, Elizabeth Majocha, and Shirley Van Nuland. 2019. Novice teacher challenges and promoting novice teacher retention in Canada. *European Journal of Teacher Education* 42, 5 (2019), 591–607. https://doi.org/10.1080/02619768.2019.1652906

[39] Azwina M Yusof and Rukaini Abdullah. 2005. The evolution of programming courses: course curriculum, students, and their performance. *ACM SIGCSE Bulletin* 37, 4 (2005), 74–78.