Experience Report: Thinkathon - Countering an "I Got It Working" Mentality with Pencil-And-Paper Exercises

Quintin Cutts, Matthew Barr, Mireilla Bikanga Ada, Peter Donaldson, Steve Draper, Jack Parkinson, Jeremy Singer, Lovisa Sundin

Goal-directed problem-solving labs can lead a student to believe that the most important achievement in a first programming course is to get programs working. This is counter to research indicating that code comprehension is an important developmental step for novice programmers. We observed this in our own CS-0 introductory programming course, and furthermore, that students weren't making the connection between code comprehension in labs and a final examination that required solutions to pencil-and-paper comprehension and writing exercises, where sound understanding of programming concepts is essential. Realising these deficiencies late in our course, we put on three 3-hour optional revision evenings just days before the exam. Based on a mastery learning philosophy, students were expected to work through a bank of around 200 pencil-and-paper exercises. By comparison with a machine-based hackathon, we called this a Thinkathon. Students completed a pre and post questionnaire about their experience of the Thinkathon. While we find that Thinkathon attendance positively influences final grades, we believe our reflection on the overall experience is of greater value. We report that: respected methods for developing code comprehension may not be enough on their own; novices must exercise their developing skills away from machines; and there are *social* learning outcomes in programming courses, currently implicit, that we should make explicit.

Introduction

In a meeting of our CS-0 programming course team, around four-fifths of the way through the semester, we were discussing a wide range of misconceptions evident among the students. The common thread across all the comments was the students' conviction that *getting the program working* was the most important consideration. Understanding either the process they'd gone through to get the program working, or the programming language constructs and patterns they'd used, was of secondary value, if any. This might not have been important if the students were going to be assessed in a lab with the support system of teaching assistants (TAs), friends and the internet all around. Instead, in just four weeks, they would face a written programming exam requiring good conceptual understanding for both code comprehension and code writing questions with nothing more than their brain and a pen in their hand.

We considered remediation. How could we enable weaker students to attempt enough exam-like questions in exam-like circumstances to give them a hope of succeeding in the final examination; in essence, to give them a flavour of Mastery Learning [2]? We discounted added traditional exercises in the last two weeks' class sessions due to insufficient preparation time for the course team. Someone suggested an all-night crash session for students "like a hackathon", with another team member extending this idea with "but for these kinds of exercises, surely it's a *Think*-athon" – and so a new concept was born. Over the next three weeks, we constructed over 200 paper-based exercises and just a few days before the exam, ran three evening Thinkathon sessions for our students.

This paper is fully an experience report. We had a course that we thought was rather well designed; as we reflected on our students' learning journey, we realised that they were insufficiently prepared for the terminal exam and so designed a remediation, the Thinkathon; as we reflected on student and staff evaluations of the Thinkathon combined with student exam performance, we developed new insights into the incorporation of code comprehension within a programming course. In particular, we find that:

- respected methods for developing code comprehension may not be enough on their own;
- students and staff who experienced the Thinkathon now recognise how important it is for novices to think and work away from machines;
- there are a number of *social* intended learning outcomes on a programming course which the instructor should incorporate and broadcast to students.

The next two sections introduce the overall course design and that of the Thinkathon respectively, making reference to the literature supporting the design choices as necessary. Section 4 outlines the evaluation methodology, with the following two sections detailing the quantitative and qualitative results of the evaluation. Section 7 discusses the whole experience, drawing out our learning, and Section 8 concludes with next steps.

Course Design

To provide context, our course is now described; one aim in doing this is to show that the development of code comprehension and conceptual understanding was not ignored in the course design. Our course, a CS-0 introductory programming course, runs over 11 weeks, with two 2-hour large class sessions and one 2-hour lab session each week. The large class sessions are based on Peer Instruction (PI) [15, 27], requiring students to complete reading and on-line tasks prior to every class, take a short quiz on that work at the start of the class, and then answer/discuss PI questions for the rest of the class.

The course design incorporates a strong code comprehension element, in line with a number of researchers' recommendations: pre-class activities encourage students to practice their code-reading ability [18]; PI questions foster understanding and analysis of programs at a range of levels of detail [4, 24]; the concept of the notional machine is introduced to learners during class, and they are shown how the activities they have been undertaking should help to develop their own notional machine understanding [6, 7, 29]; labs encourage the development of a number of smaller programs rather than just one large one [10]; some lab sessions encourage students to explore complete programs and draw out re-usable patterns [19]; the staff-student ratio in the labs is high to provide sufficient time for students to talk to staff about their programs, displaying their understanding. These discussions are intended to emphasise the importance of understanding over simply getting programs working. TAs are expected to quiz the students about their programs, for example, pointing at a piece of code and asking the student to explain how it works and why they used it, and award an advisory progress grade depending on the students' responses.

In line with other courses [9, 26], this course introduces students to two languages: for the first 5 weeks, students learn a block-based language, in this case Alice; for the remaining 6 weeks, they learn Python. To give a sense of the scope of the course, the most complicated problems explored will require reading in data from a text file, storing it in a data structure requiring some combination of lists and dictionaries, processing the data structure in some way usually according to user input, and finally writing results either to screen or back into a file.

While the course does have a practical assessment for credit, the primary assessment is a written final examination. The questions in this exam require students to display both code comprehension and code writing skills, with no access to notes or a machine. The course designers believe that the questions are simple enough that a competent hard-working student should be able to internalise the programming concepts, syntax and patterns during the course, and therefore should not require recourse to external aids. The exam contains the following four question types:

- Evaluation of expressions involving simple types and lists and dictionaries
- Hand/desk execution
- Identifying and correcting errors in code, given a problem statement
- Problem-solving: given a problem statement, write a solution in programming language code

Thinkathon Design

The Thinkathon idea was born from the realisation that students were not developing sound conceptual understanding, and we aimed to provide an extended immersive experience – not unlike a "boot camp" [30] – to correct this. The Thinkathon design was iterative, involving discussion among our 17-strong CS education group, including PhD students and academic staff, mainly CS but with some members from other schools, with widely varying breadths of experience. Our overall goals were to:

- enable the students to succeed in the upcoming exam and be well-prepared for follow-on programming courses;
- help them to develop appropriate attitudes and approaches to learning programming, for example, to have the confidence to know whether or not they've correctly solved a problem, tackled a bug, or written a fragment of code.

The issues with the students' learning that arose in our discussions and that we were aiming to address were perceived to be as follows:

- The operation of programming language constructs is not well-enough understood by students, as evidenced by the poor quality of discussions between TAs and students in labs. The use of PI in CS is typically expected to address this [27], but we are not the only ones to find that PI does not ensure all students succeed [12]. We suspect that while the preparation and discussion aspects of PI do ensure students understand concepts *in the moment*, there is not enough repetition for these understandings to become permanent.

- The number of larger problems/tasks and solutions developed or studied by students is not enough. PI cannot easily handle this issue, as PI questions typically deal with small code fragments. This results in the students not developing a sufficiently advanced pattern memory, without which they are unable to quickly break down a new problem into known chunks with known solutions.
- The students are too dependent on outside sources of help, preventing them from developing their own understanding and confidence. For example, the programming environment is used as a crutch to tell them whether what they've developed is correct; the internet can be used to find fragments of code to solve problems, a practice rarely appropriate in an introductory programming lab, unless the student knows that he/she has to thoroughly understand such fragments; and, help from friends or TAs may simply solve a student's programming problem, rather than helping them learn how to solve their own problem, a more useful long-term goal.

Our task was to develop exercises for the students to tackle on the Thinkathon evenings, completion of which would help to meet the goals and address the issues identified above. We came up with five objectives for the Thinkathon exercises:

- All the exercises should be paper-based to break the dependence on a machine. Such "unplugged" computing activities [1] are generally deployed for younger learners and not at university level.
- There should be sufficient exercises of every type, along with model solutions, to support Mastery Learning, identified as an under-used but highly desirable learning practice for programming education in [14, 25].Under such a practice, a student should be able to find out if their solution attempt is correct; if it is not, to work out why, and then to try again on another similar exercise; if it is correct, and the student has now completed a number of similar exercises correctly, then they can move on to exercises at the next level of difficulty.
- Students should be able to work at their own pace, essential for independent learning and in a subject where students are at so many different levels.
- Students should not get stuck often. This requires the exercises to follow a developmental sequence, where each incremental step is small. The development of such sequences is not regarded to be straight-forward in CS [21].
- Students should be able to gain assistance from other students, but not to be dependent on it. A student's peers are a strong support mechanism, and it is important to develop those connections, but also to make sure that students are aware of more and less constructive ways of supporting their peers.

We struggled with the complexity of large numbers of each type of exercise for Mastery Learning, combined with the two dimensions associated with developing a developmental sequence. First, there is a progression from code comprehension to code writing – e.g. it is easier to trace code than it is to write from scratch [13]; second there is a progression through examples that make use of steadily more complex, and more complex combinations of, programming language constructs and programming patterns.

We judged there to be no perfect progression that we could readily develop, and so made a compromise. We created 5 packs of exercises, sequenced on the comprehension/writing dimension, containing several kinds of exercises, which largely match the style of exam questions for the course. The exercise styles, described with reference to what aspect of Schulte's Block Model for code comprehension [24] must be understood to complete them, are:

1. Describing, evaluating and creating expressions and simple statements -- an extension of our exam practice to ensure that students thoroughly understand primitives. Structure section of the Block Model, at the Atom level only.
2. Hand/desk execution. Block Model's complete Structure section.
3. Parsons Puzzles [20]. These code rearrangement exercises represent a developmental stepping-stone from code reading to writing. All parts of the Block Model, both Structure and Function.
4. Validating/debugging code. These exercises have the problem statement and some incorrect code, and the instruction to find and fix the errors. All parts of the Block Model.
5. Problem-solving from scratch

Broadly, the exercises in each pack were ordered to follow a typical sequence for introducing programming language concepts and patterns. Novice programmers value exposure to example programs [22]. Our packs provided sequences of such programs in a scaffolded setting. "Learning by doing" is considered by novices to be effective; they report exercises to be more useful than lectures [11]. Here, students started with code completion activities [17], moving through the recognition of plans and patterns [28] and finishing with coding *ex nihilo* problems [16].

Students were alerted via email and lecture to the Thinkathon sessions, which occurred immediately prior to exams. The sessions ran from 4-7pm, with pizza and soft drinks provided half-way through, initially on two consecutive evenings. The students encouraged us to run a third session. At each session, around 8 TAs/instructors were on hand to check student work and answer queries. The packs were only available on paper. Students were directed to use the materials as follows: start at the first exercise pack; dip in at any point; if the questions seemed easy, jump ahead, if hard, then drop back; once the exercises are complete, take them to a TA to be checked, and if they're correct, move on to the next pack.

In the interests of fairness, all the exercises and their answers were uploaded to the course VLE (Moodle) after the third session. This enabled access to those students who didn't attend, citing pressure of revision for exams scheduled earlier than the CS exam, and students who hadn't completed all packs by the end of the three sessions.

The materials can be found at: http://ccse.ac.uk/thinkathon.

Evaluation Methodology

As part of the Thinkathon, we developed and implemented an evaluation to establish whether students' academic performance improved after the introduction of the new practice, and to determine student and staff views.

The intervention, which sought to deepen the understanding of programming among learners in line with code comprehension ideals and to explore the impact of a change in practice, used action research as a means of enquiry. Action research involves “practitioners studying their own professional practice and framing their own questions. This style of research has the immediate goal to assess, develop or improve practice” [31].

We were interested in the following questions:

1. What are the students’ perceptions of the Thinkathon exercises?
2. Is there a difference in performance in the final exam between Thinkathon participants and those who did not attend, possibly taking into consideration mid-term exam performance.
3. Do those who attended the Thinkathon perform better than those who only accessed the questions and answers online.
4. Does the Thinkathon highlight deficiencies in the current course design and if so how?

The qualitative evaluation was designed to address the first and fourth questions, with the quantitative evaluation addressing the second and third.

The study participants were students enrolled in our CS-0 course. There was a total of 114 students. However, only 108 took the final exam. Ethical approval was obtained before the study. To compare the group who attended the Thinkathon to those who did not participate, the study used their mid-term and final exam scores. To analyse student perception of the Thinkathon exercises, a survey was sent to students before and after they had attended the Thinkathon. Finally, a round-table meeting of all staff involved in the Thinkathon was held where we collected staff viewpoints, sometimes recording verbatim quotes and sometimes a summary of particular views.

Descriptive and statistical analyses of means and standard deviation were calculated to provide quantitative results. A comparative examination of participants’ confidence level before and after the Thinkathon was also performed. Qualitative data from student Thinkathon participants was collected using four open-ended questions in the post-test survey.

- Reflecting on the Thinkathon exercises, what programming concepts or types of problem did you initially find difficult?
- In what ways do you feel the Thinkathon sessions helped you better understand these concepts or problem types?
- What else - not covered above - was good about the Thinkathon session(s) you attended? What other realisations or learning or gains about programming have you made as a result of attending?
- What could we have done better at the Thinkathon session(s) you attended?

In the initial coding pass, an inductive analysis of individual student responses to survey questions was carried out independently by two of the paper’s authors. However, during comparison and discussion of these initial results it became apparent that a simultaneous coding approach [23] using a hierarchical coding scheme [23] that grouped all responses by student would be more appropriate; the same issue could appear across several of their answers, with distinct aspects of interest like the instructional design and their own level of understanding blended together. This was supplemented by a simpler inductive descriptive coding [23] of a summary of the Thinkathon staff debriefing meeting.

Quantitive Results

For an understanding into the Thinkathon’s effect on the students’ Python skills, attendees’ survey data was analysed in combination with their final exam performance.

Out of the 108 students who took the final exam, 37 took the pre-test survey, and 34 took the post-test survey. From the union of these responses, it can be estimated that 44 students participated in the Thinkathon event while 64 didn’t.

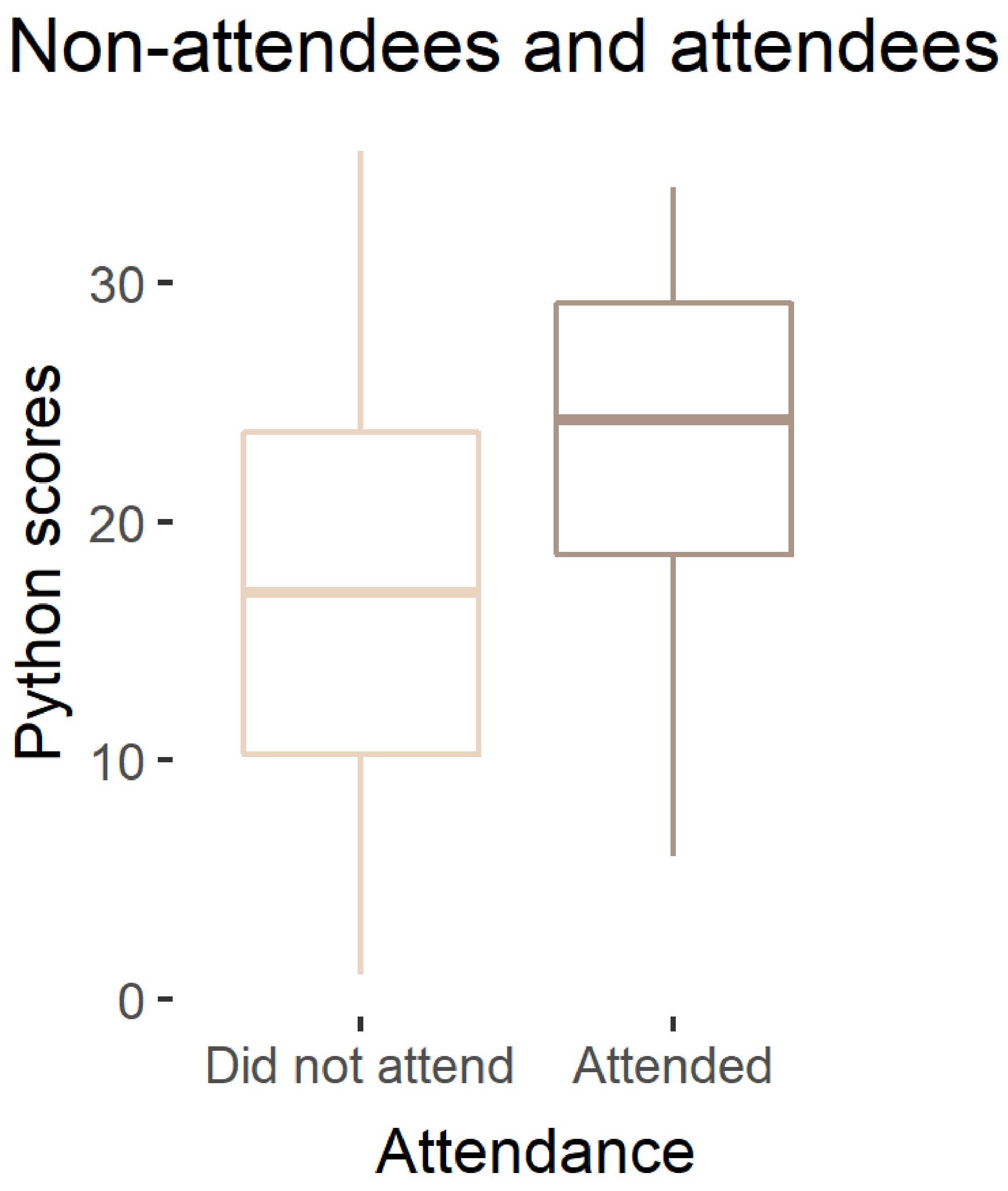

Figure 1: Final exam scores as boxplot

First, we looked into whether there was a group difference among attendees and non-attendees for the first two Python exam questions, since they were concerned with the first set of Thinkathon exercises, which attendees were most likely to have completed, namely hand execution. Descriptive statistics indicated no such differentiating ability. The full set of Python questions did, however, reveal that attendees on average scored higher (see the Figure 1). Attendees had a mean of 23.42 (SD = 7.5) while non-attendees had a mean of 17.37 (SD = 8.31), with a maximum score of 37.

This inference had to be qualified by the lack of random allocation of students to attend/non-attend groups. Those that chose to attend may already have had greater experience, academic conscientiousness or test-taking ability. We reasoned that students' mid-term performance could be used as a proxy for these, since this shows a linear association with the final exam (see Figure 2); yet, looking into the ranks in the mid-term versus final exam, Thinkathon attendees had a median improvement of 7.5 places while non-attendees on average ranked 6.5 places *lower*.

Thus, a one-way analysis of covariance (ANCOVA) was conducted. Even after controlling for mid-term performance, there was a statistical difference on performance by attendance [F(2, 104) = 29.76, p<.05]. This suggests that the Thinkathon contributed causally to an improvement in Python programming ability.

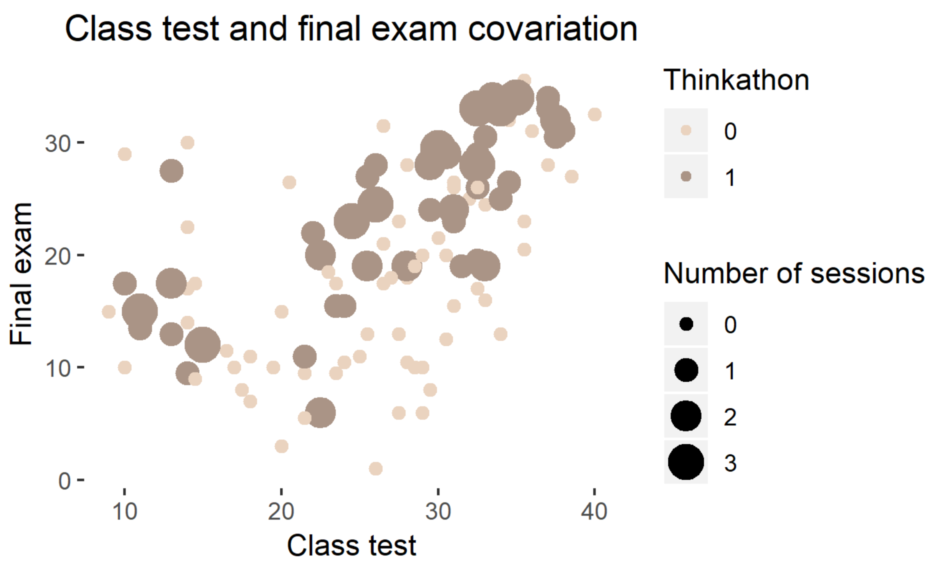

Figure 2: Relationship between mid-term result and final exam

This is a conservative analysis: while attendance was based on completion of the on-line surveys and not on a session register, we counted over 50 attendees on the first evening, and so some students must have not completed the survey; also, 32 people downloaded the packs after the event without attending. These aspects would tend to flatten any correlation. Descriptive analysis suggested that "downloading only" led to no discernible improvement compared with doing neither – only the group that also attended improved.

In an analysis of the 23 participants who completed both the pre and post survey we found there was a modest increase in confidence about the exam from a mean of 2.91 (SD = 0.95) to a mean of 3.30 (SD = 0.63). When comparing all pre or post survey participants (n=42) who had made final grade estimates to their actual grades we discovered that the upper-quartile of final exam performers most frequently underestimated their final exam grade and the lower-quartile most frequently overestimated their final grade.

Qualitative Results

Student Experience

Two overarching themes emerged from qualitative analysis of the post-Thinkathon survey data: *instructional design* and *student learning*. Most of the discussion relating to the instructional design either highlighted benefits or improvements that could be made either to the course or future Thinkathon events. Student learning comments mostly related to either perceived benefits of taking part or particular difficulties that they realised they had.

Reflecting on the instructional design of the Thinkathon, student responses related primarily to the support afforded by tutors and peers, the nature of the presented exercises, and the opportunity to practice solving problems. Tutors were variously described as being "super engaged", "helpful and supportive", and "very encouraging", while peers were thought to provide a useful alternative perspective: "Having not just tutors but other students helped, as we could discuss problems and by seeing things slightly differently helped me to understand different problems and how to solve them". The literature tells us novices benefit from interaction with their peers [5]. They also maintain high motivation to succeed when they receive personal tutoring from expert humans [3]. The informal Thinkathon environment fostered both these kinds of interactions.

The way the exercises were presented proved more divisive, with some students appreciating that the high number of similar exercises provided an opportunity to "practice them over and over", while others felt the exercises to be "repetitive". One student summarised this tension as follows: "whilst it was good there was so many questions in each booklet, it gave a lot of practice, at the same time I think it's very easy to get bogged down in them". Students also suggested that a greater sense of variety might be achieved by enforcing some system of rotation, whereby groups of students would be tasked with completing one kind of exercise before being asked to move to another.

The value of practice was frequently expressed in terms of the upcoming examination, with comments relating to "writing code on paper" or how the exercises "helped me see how exam questions would be laid out".

Student participants described their learning experience primarily in computing science-related terms, commenting on how the exercises enhanced their ability to comprehend and write code. The experience also appeared to facilitate learning that might be described as affective (improving confidence), cognitive (enhancing problem-solving skills), or meta-cognitive (facilitating self-assessment).

Improvements to code comprehension were described in terms of understanding the code at hand ("trying to solve or execute a code without a computer helps for deeper understanding"), including the attendant concepts, and being able to describe it ("describing code showed me what terminology I need to revise in order to talk freely about code").

References to the Thinkathon experience leading to improved code writing ability were abundant, and the challenge of being asked to write programs "from scratch" was highlighted by many as being most significant: "It [Thinkathon] has helped me really think about a program before writing it. Before I'd just write anything that came to mind and fix it until it worked but now I tend to get correct solutions the first or second attempt at a problem."

In terms of self-assessment, the following comment proved particularly illuminating: "I didn't know what to expect on the exam. In a way, I didn't know what I didn't know." This echos findings that students with low or partial knowledge find it difficult to accurately assess their own level of competence and tend to overestimate their ability [8].

Staff Experience

The one-and-a-half-hour discussion among the 17 TAs and staff who had been involved in either or both of the Thinkathon design and the sessions themselves elicited a number of interesting topics.

Those present at the sessions could tell how much the students enjoyed them, and were surprised at the high level of interaction between students – they were helping each other a lot. The first two packs clearly got the students thinking: a TA said "hand execution forces you to think through code, to think about every line – e.g. mental models of variables. Can get lazy in front of a machine – try, try, try until it works." Indeed, the first pack which included expression evaluation, naming constructs and describing how they operated, as well as expression writing, took most students the whole of the first session. This was a great surprise to staff. When queried, students were keen to stress that they did not want to miss out on anything and to make sure they had a firm foundation. Staff described seeing real "light-bulb moments" for the students, e.g. that a single element slice of a list is still a list, and understanding fully about the identity of data structures and their mutability – aspects discussed in class but perhaps not followed up enough.

The high level of interaction was somewhat at odds with our intention to help the students develop the confidence to be able to answer questions on their own. In discussion with staff, the students felt the peer interaction was essential, and repeatedly stressing our goal for them to answer confidently on their own did not change their behaviour. Students noted how confronting it could be "to talk to folk I don't know well about what I don't know", indicating their nervousness in approaching instructors and even TAs. The implication here is that the Thinkathon questions highlighted for the students that there was a lot they didn't know; and that for most students, not just a few, there was real learning even at the simpler levels. This was not just a revision session.

Considering self-direction in the students' study, we discussed the relationship between the Thinkathon exercise packs and a text-book in, say, mathematics that has a very large number of questions, with answers, presented in a well-tried developmental sequence. While staff remembered using such text-books in a self-study Mastery Learning manner in mathematics and other subjects, they also recognised that larger programming problems did not have the kind of direct right/wrong answers as had the math questions.

Those in our group from other disciplines commented on how labs are quite disconnected from lectures in the other sciences, and exam preparation is based around reading of notes, hand-outs and a text-book. In computing, the labs look real, authentic. By comparison, neither the Thinkathon nor the exam look realistic. A non-CS colleague commented "getting them to do stuff on paper exams is the wrong thing", a prevalent attitude among computer scientists too, but another responded "the exam is doing exactly the right thing, getting the student to articulate what they do and don't know, which is not required or effectively evidenced in the lab."

This highlights the challenges we have in effectively signposting to students the learning outcomes of a programming course.

Discussion

While we have been excited and motivated by the Thinkathon experience, attendance appearing to raise final course grade, it has also delivered a salutary lesson. We started off acting on a clear misunderstanding in students' minds between lab activities and the written exam, but we have realised that the problem lies somewhere in the learning design between the PI-based lectures and the lab sessions. Furthermore, the students really don't know what it is they should be learning. Unpacking this, we note the following.

We have depended on PI to satisfy the code comprehension side of our curriculum. The pre-lecture work should help to embed concepts; the discussion in class should induct novices into a programmers' community of practice. Coupling this with relatively traditional lab-based practical code writing activities has proven to be wanting, however. The evaluation tells us that students, including good students, don't know even what we might consider the basics: naming of constructs, explanation of how they work, evaluation of expressions, simple desk execution. They spent an entire evening on this material. Fundamentally, we have forgotten how much there is to know, taking it for granted. Why should a novice be expected to pick up a new vocabulary for a new kind of language and the grammatical terms required to be able to talk about it, without explicit instruction? PI does try to address this, but we suspect it does it in too inefficient a manner.

On reflection, both staff and students now realise the huge value of thinking and working on programming exercises *away from machines*. We will re-introduce tutorial sessions in rooms away from our labs, in which students work on Thinkathon style problems, and possibly with regular hand-ins and grading. We moved away from these in years past, for logistical reasons and also to maximise programming time – but that was before we understood about the importance of building the foundations soundly.

Tutorials bring with them the notion of discussion, and this brings to mind Francis Bacon's instruction from the 1600s that a good education required both reading and writing, and also "conference" – the ability to speak and argue one's case. We have writing in abundance in our programming courses, and code comprehension indicates that we should have reading, but what of conference? If we can't talk and argue about our practice, what kind of practitioner are we? PI again can support this, but are we doing enough to foster fruitful discussion, and have we identified good ways of putting it to use? Our own labs have included limited elements of discussion, but we could add code reviews or set up discussions on the benefits of different programs solving the same problem.

The overall experience suggests to us a number of "social" learning outcomes for a programming course that currently lie hidden:

- The students need to know the full range of what matters to us – explaining code using the correct language, hand-execution, developing solutions away from machines (and we need to adjust our instructional designs accordingly)

- Students must be able do all this *on their own*, and do it exactly. Furthermore, with practice, we expect them to be able to complete these activities correctly *first-time*. It is no good for their peers or the TA to show them the answer.
- A resource like the Thinkathon packs are *for individuals*, not a group thing. It is a self-learning resource at best, enabling students to find out what they know and what they don't.

The Thinkathon succeeded on the first of these points, but even with our encouragement to the students during the sessions, it was less effective on the second two points.

Conclusion

Realising late in our CS-0 course that our students were not prepared for the upcoming written exam, we developed an evening revision session format we dubbed *Thinkathon*, that proved effective in raising students' grades. Evaluating the exercise has led us to identify a number of important short-comings in our PI-based course design, and consequent adjustments. In particular, we note: the need to learn language vocabulary and grammatical terms thoroughly; the value of working away from machines; the importance of discussion; and the explicit presentation of *social* learning outcomes for a programming course.

Acknowledgements

Many thanks to those who spent hours in discussion, developing the Thinkathon exercises and supporting students during the sessions: Niall Barr, Elizabeth Cole, Sarah Honeychurch, Fionnuala Johnson, Derek Somerville and Ethel Tshukudu.

References

1. Timothy C. Bell, Mike Fellows, and Ian H. Witten. 1998. Computer Science Unplugged: Off-line Activities and Games for All Ages. Computer Science Unplugged.
2. Benjamin S Bloom. 1968. Learning for Mastery. Instruction and Curriculum. Regional Education Laboratory for the Carolinas and Virginia, Topical Papers and Reprints, Number 1. Evaluation Comment 1, 2 (1968).
3. Kristy Elizabeth Boyer, Robert Phillips, Michael D. Wallis, Mladen A. Vouk, and James C. Lester. 2009. Investigating the role of student motivation in computer science education through one-on-one tutoring. Computer Science Education 19, 2 (Jun 2009), 111–135. https://doi.org/10.1080/08993400902937584
4. Quintin Cutts, Sarah Esper, Marlena Fecho, Stephen R. Foster, and Beth Simon. 2012. The Abstraction Transition Taxonomy: Developing Desired Learning Outcomes Through the Lens of Situated Cognition. In Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12). ACM, New York, NY, USA, 63–70. https://doi.org/10.1145/2361276.2361290
5. Adrian Devey and Angela Carbone. 2011. Helping First Year Novice Programming Students PASS. In Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114 (ACE '11). Australian Computer Society, Inc., 135–144. http://dl.acm.org/citation.cfm?id=2459936.2459953

6. Benedict du Boulay. 1986. Some Difficulties of Learning to Program. Journal of Educational Computing Research 2, 1 (1986), 57–73.
7. Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. International Journal of Man-Machine Studies 14, 3 (1981), 237–249. https://doi.org/ttps://doi.org/10. 1016/S0020-7373(81)80056-9
8. David Dunning, Kerri Johnson, Joyce Ehrlinger, and Justin Kruger. 2003. Why People Fail to Recognize Their Own Incompetence. Current Directions in Psychological Science 12, 3 (2003), 83–87. https://doi.org/10.1111/1467-8721.01235 arXiv:https://doi.org/10.1111/1467-8721.01235
9. Dan Garcia, Brian Harvey, and Tiffany Barnes. 2015. The Beauty and Joy of Computing. ACM Inroads 6, 4 (Nov. 2015), 71–79. https://doi.org/10.1145/2835184
10. Kelly Downey Joe Michael Allen, Frank Vahid and Alex Daniel Edgcomb. 2018. Weekly Programs in a CS1 Class: Experiences with Auto-graded Many-small Programs (MSP). In 2018 ASEE Annual Conference & Exposition. ASEE Conferences, Salt Lake City, Utah.
11. Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Jarvinen. 2005. A Study of the Difficulties of Novice Programmers. In Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05). ACM, 14–18. https://doi.org/10.1145/1067445.1067453
12. Soohyun Nam Liao, Daniel Zingaro, Kevin Thai, Christine Alvarado, William G. Griswold, and Leo Porter. 2019. A Robust Machine Learning Technique to Predict Low-performing Students. ACM Trans. Comput. Educ. 19, 3, Article 18 (Jan. 2019), 19 pages. https://doi.org/10.1145/3277569
13. 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. ACM, 101–112.
14. 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 (ITiCSE 2018 Companion). ACM, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779
15. Eric Mazur. 2017. Peer Instruction. Springer Berlin Heidelberg, Berlin, Heidelberg, 9–19. https://doi.org/10.1007/978-3-662-54377-1_2
16. Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '01). ACM, 125–180. https://doi.org/10.1145/572133.572137 Canterbury, UK.
17. J. J. G. van Merrienboer and H. P. M. Krammer. 1990. The "completion strategy" in programming instruction: theoretical and empirical support. Research on instructional design and effects / eds. S. Dijkstra, B.H.A.M. van Hout Wolters, P.C. van der Sijde (1990), 45–61.
18. Heng Ngee Mok. 2014. Teaching tip: The flipped classroom. Journal of Information Systems Education 25, 1 (2014), 7.

19. Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '07). ACM, New York, NY, USA, 151–155. https://doi.org/10.1145/1268784.1268830
20. Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In Proceedings of the 8th Australasian Conference on Computing Education-Volume 52. Australian Computer Society, Inc., 157–163.
21. Anthony Robins. 2010. Learning edge momentum: A new account of outcomes in CS1. Computer Science Education 20, 1 (2010), 37–71. https://doi.org/10.1080/ 08993401003612167
22. Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. Computer Science Education 13, 2 (Jun 2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200
23. Johnny Saldaña. 2015. The coding manual for qualitative researchers. Sage.
24. Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching. In Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08). ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535
25. P. Seeling. 2016. Switching to blend-Ed: Effects of replacing the textbook with the browser in an introductory computer programming course. In 2016 IEEE Frontiers in Education Conference (FIE). 1–5. https://doi.org/10.1109/FIE.2016.7757620
26. Beth Simon and Quintin Cutts. 2012. How to Implement a Peer Instructiondesigned CS Principles Course. ACM Inroads 3, 2 (June 2012), 72–74. https: //doi.org/10.1145/2189835.2189858
27. B. Simon, M. Kohanfars, J. Lee, K. Tamayo, and Q. Cutts. 2010. Experience report: peer instruction in introductory computing. , 341–345 pages. https: //doi.org/10.1145/1734263.1734381
28. Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. IEEE Trans. Softw. Eng. 10, 5 (Sep 1984), 595–609. https://doi.org/10.1109/ TSE.1984.5010283
29. Juha Sorva. 2013. Notional Machines and Introductory Programming Education. Trans. Comput. Educ. 13, 2, Article 8 (July 2013), 31 pages. https://doi.org/10. 1145/2483710.2483713
30. John Stamey and Steve Sheel. 2010. A Boot Camp Approach to Learning Programming in a CS0 Course. J. Comput. Sci. Coll. 25, 5 (May 2010), 34–40.
31. Jane Zeni. 1998. A guide to ethical issues and action research [1]. Educational action research 6, 1 (1998), 9–19. https://doi.org/10.1080/09650799800200053