An Assignment a Day Scaffolded Learning Approach for Teaching Introductory Computer Programming

Deepak Dawar
daward@miamioh.edu

Marianne Murphy
murph103@miamioh.edu

Miami University
Hamilton, Ohio

Abstract

Teaching introductory programming courses to university students who come from a varied set of academic and non-academic backgrounds is challenging. Students who are learning programming for the first time can become easily discouraged leading to procrastination that subsequently can have an unfavorable effect on their learning outcomes, and overall final grade. This work proposes An Assignment A Day (AAAD) Scaffolded Learning approach, and presents our experiences with this pedagogical approach. According to neuroscience research, when subjects are engaged continuously with a task, there is improvement in the brain's neuroplasticity. Based on this research and our own experiences with entry level programming students, we pursued the research question: "Can a targeted continuous engagement with course material, and problem solving assignments improve learning outcomes?" The students, instead of writing an assignment and a lab for each module, were asked to complete one assignment a day, not exceeding four assignments a week. The limited areas of impact that we targeted were student procrastination in submitting assignments, student failure to submit assignments, and student engagement. The overall acceptance of this technique by students has been quite positive, and we report an improvement in assignment submission rates, and final exam scores, apart from improved student engagement. Students found the approach extremely effective in spite of having to spend considerable amount of time on assignments almost everyday.

Keywords: Introductory level programming, pedagogy, student engagement, neuroplasticity, student procrastination, learned helplessness.

1. INTRODUCTION

Introductory programming is an arduous process for many students especially those who have little or no prior experience. Low course completion rates are consistently reported (Bennedsen & Caspersen, 2007; Newman, Gatward, & Poppleton, 1970; Allan & Kolesar, 1997; (Sheard, & Hagan, 1998; Beaubouef & Mason, 2005; Kinnunen & Malmi 2006; Howles, 2009; Mendes et al., 2012; Watson & Li, 2014). Apart from learning and recognizing the syntax and semantics of the programming language, one also has to create a mental model of the solution (Sorva, 2013). The novice programmer has to grapple with multiple domains of learning as suggested in the literature (Rogalski & Samurçay, 1990; Kim & Lerch, 1997; Robins, Rountree, & Rountree, 2003; Davies, 1993).

It has also been suggested that the most difficult aspect faced by novice programmers may not be related to the specifics of the language at all. According to Lahtinen, Ala-Mutka, & Järvinen, 2005, understanding how to design a program,

and dividing functionality into procedures are the primary problems faced by entry level programming students. Further, even after successful course completion, student learning in these introductory programming courses is not always retained (McCracken et al., 2001; Utting et al., 2013). Does that mean that programming as a course is more difficult than other similar level courses? There is no consensus on this theory, but there is a large body of data to suggest that this might be the case (Luxton-Reilly, 2016). In-fact, when computing courses were studied under the framework of two prominent taxonomies i.e. SOLO (Brabrand & Dahl, 2009), and BLOOM (Oliver et al., 2004) these courses were found to be more challenging than other courses. A recent study by Margulieux, Catrambone & Schaeff er.,2018, compared the domain difficulty of three courses – computer programming, chemistry, and statistics, and found computer programming to be the most difficult of three due to the complexity of the content to be learned and handled at a given time.

The authors of this paper have faced similar challenges in their classrooms while teaching introductory programming classes. From less than desirable passing rates, to inability of students to apply the learned concepts in subsequent programming classes led us to investigate the reasons more closely as relevant to our classroom setup, and provide possible interventions and remedies. This work is the result of one such intervention. The authors observed that one of the primary reasons for learning outcome failures in the class was student's procrastination and lack of motivation to finish the assignment(s) on time. Motivation is a vast subject in its own right, and can take myriad forms.

We suspect that the lack of motivation and procrastination may just be symptoms of an abnormal cognitive load that programming assignments, and related tasks carry for many students. Cognitive load theory (Sweller, 1988, 1994; Paas, Renal, & Sweller, 2003; Plass, Moreno, & Brünken, 2010) deals with the aspects of load placed on working memory while a task is being executed. The amount and nature of this load depends upon the interactive nature of elements involved in the tasks. Computer programming requires balancing numerous interactive tasks. For example, writing a computer program involves juggling numerous details like problem domain, current state of program, language syntax, strategies etc. (Winslow, 1996).

The landscape of the potential problems faced by novice programmers is vast, and is quite formidable. Instead of dealing with the motivational aspect of programming directly, we turned to an approach that couples program scaffolding with the generally accepted notion that constant practice improves the learning outcomes, and as shown by psychological studies (Brown & Bennett, 2002; Moors & De Houwer, 2006; Glover, Ronning, & Bruning, 1990) done on variable student populations. Constant practice can also make students want to learn more (Kalchman, Moss, & Case, 2001). Constant practice and improved problem solving skills have shown to be mutually dependent and shown to be in a complex relationship as shown by Eckerdal, 2009. There is a plethora of studies confirming the important role practice and experience play in developing problem solving strategies by novice programmers. In a series of studies conducted by Rist (1986, 1989, 1995, 2004), and reviewed by Sorva (2012) confirm that one of the main differentiator of students into novice and expert programmers is their constant engagement and experience with learned schemata.

Keeping these factors in mind, we designed An Assignment A Day (AAAD) Scaffolded Learning approach wherein students were given a programming assignment a day, and no more than four assignments a week. Every assignment built on the previous assignment(s), and the final assignment was to be a mini-project testing students on all the concepts learned so far in previous assignments. We are faced with a few dilemmas though. First, it has been shown that constant testing of students leads to high levels of anxiety that may lead to sub-optimal performance (Kaplan et. al, 2005). Second, solving hard problems can easily bring down the morale of the novice programmers, and may send them into the spiral of learned helplessness, leading to poor performance (Crego et. al, 2016). To mitigate these effects, and at the same time make the students practice as much as possible, we made sure that the opening assignment tests very basic concepts, and then subsequent assignments gradually increase in complexity. We opined that having assignments designed in increasing order of complexity will reduce cognitive load on students thereby possibly resulting in better learning outcomes.

This opinion was based, in part, on classroom observations, and a study conducted by Alexandron et al. (2014). This study demonstrated the effectiveness of aligning tasks in increasing order of complexity on cognitive

load, though the mandate of the study was much wider than studying this correlation.

2. METHODOLOGY

We created An Assignment A Day (AAAD) Scaffolded programming approach for introductory programming courses for our student population. The main driver of this intervention was the observation that in the orthodox model (one assignment a module that we followed), many students tend to procrastinate, and delay working on the assignments as late as possible. When the submission deadline approaches, they jump into action. It is evidenced from our experience that quite a high number of questions from students are received in last three hours prior to submission deadline. They are then faced with multiple complexities of the assignment leading to increased cognitive load. This increased load may give rise to student frustration, unwillingness to continue to work on the assignment, and eventually may lead to unfavorable learning outcomes. The purpose of this intervention was to make students constantly practice the material thereby potentially improving their chances of learning the material. We opined that this approach will assert a slight positive stress on students to submit the assignment at the end of the day. We also realized that the possible success of this scheme will significantly depend upon rendering the cognitive load asserted by the assignments, germane or manageable. AAAD was designed keeping all these possibilities in mind.

Our method is quite simple − make the students practice constantly and assert just the optimum stress on them in terms of deadlines and materials, so as to avoid student disenchantment and frustration with the course, while simultaneously improving learning gains. Being run for the first time, and due to small sample size, we are not in a position to define as to what constitutes the optimal load, as of now.

We tried to keep the AAAD approach as straightforward as possible with a few exceptions in between. The approach can be summarized as:

1. Students will ideally do one assignment per day
2. Opening assignments of the chapter will test students on very basic skills like writing a method stub. Subsequent assignments will gradually increase in complexity keeping in mind the cognitive load asserted by the assignment
3. There will not be more than four assignments per week
4. The final assignment should test students on all the previously learned chapter concepts
5. As an exception, and depending upon the cognitive load, an assignment may be completed in two or more days rather than a single day. This should mainly apply to the last assignment of the chapter that tests students on multiple concepts, but can also extend to assignments that variably test single but difficult concepts, and are not the last assignment of the chapter.
6. All other factors like quizzes, projects etc. remain the same for experimental and the control group.

The study was conducted over two semesters. The control group data was collected in the first semester. This group worked with the orthodox approach followed at our institution for introductory programming classes i.e. on an average, one assignment and one lab per week, with quizzes at the end of the module/chapter.

In the next semester, the experimental group was administered the AAAD approach, and data collected at the end of semester. A total of 37 assignments were given to the experimental group over a course of 13 weeks of which 1 week was spring break. Rest of the 12 weeks meant 84 days of which weekends accounted for 24 days. 10 days were meant for quizzes and exams. Hence, the students had to complete 37 assignments in about 50 days i.e. about 0.75 assignments a day. An additional end of course survey was conducted with the experimental group to measure how well this approach was received by the students.

Student Population

The student population of our department consists of both traditional and non-traditional students, though the terms are not well defined in literature. For the purposes of this work, we define traditional as students who are full time, and are recent high school graduates. Non-traditional students are those who have full-time jobs, are part-time students, and/or are older, and seeking a new career for a variety of reasons.

The number of students in the control and the experimental group were 20 and 22 respectively. One student from the control group declined to have their data included in the study. The course is mandatory for Computer Science (CSE) students but can be used as an elective

for Information Technology (IT) majors. The control group had 12 IT/CSE majors and 8 non IT/CSE students. The experimental group had 13 IT/CSE, and 9 non IT/CSE majors. So the class composition of both groups is fairly similar, with the control group and experimental group having about 40% and 41% non IT/CSE majors respectively. This relatively similar class composition gives us some confidence about the experimental set up. It could have been quite difficult to compare results, had the IT/CSE and non IT/CSE major ratios varied widely.

Sample Load

To describe the procedure effectively, a sample load is presented here. The chapter/module to be presented is "method writing" in JAVA. This was to be delivered as an eight-day module with classroom practice labs (non-graded), five assignments, and a quiz at the end. Here is brief a description of assignments. Detailed descriptions of these assignments are included in Appendix B. As can be seen from Table 1 (see Appendix A), even a slight modification of problem statement can quickly increase the number of concepts that the student has to deal with, thereby increasing the cognitive load. This issue, in our opinion has to be dealt with effectively, if we are to improve upon the chances of student learning.

Comparison

Since the experimental group had to do many more assignments (at least 4 more assignments per module), an equitable comparison between the groups was a challenge. We decided that the comparison of the last summative assignment given to the experimental group with the usual assignment given to the control group would make a fair comparison. Both these assignments were similar in terms of concepts they tested but there were also some differences. For example, they differed in cognitive load and total points in many cases. The experimental group students have had more exposure to the concepts since they would have submitted a series of assignments by now. Our intervention assessed the following metrics for both groups, and for each assignment compared.

- Late submissions
- No submissions

To measure the impact of our technique on overall grades, if any, we administered the exact same module quizzes, and final exam to both groups and compared the following data points for both groups:

- Module wise quiz scores
- Final exam scores

Apart from this inter group comparison; we also performed an intra-group comparison for the experimental group to track student performance within the module, and the course as a whole. Observations and results are listed, and analyzed in next section.

3. RESULTS

We divided our analyses into two parts - inter and intra group. Inter group analyses compared the control with the experimental group, and intra group analyzed just the experimental group.

Inter Group Analyses

The control group did only one assignment per week whereas the experimental group did several leading up to the last assignment of the module. We compared the statistics of the last module assignment with the usual assignment of the control group. As an example, for assignments listed in Table 1, in the control group, an assignment similar to 5 was given to the students. In the experimental group, however, the same assignment 5 was given as the last assignment, after students have had some exposure to the relevant concepts in the previous assignments vis-à-vis assignments 1, 2, 3, and 4.

Table 2, 3 and 4 summarize the data points collected for comparison. The number of possible submissions per module in the control and experimental groups were 20 and 22 respectively.

No Submissions		
Module	Control (20)	Experimental (22)
1	1	0
2	0	0
3	0	0
4	2	0
5	2	1
6	5	3
7	4	3
Total	14	7

Table 1: Assignments not submitted per module

Late Submissions		
Module	Control (20)	Experimental (22)
1	0	1
2	1	2
3	1	3
4	1	2
5	1	5
6	4	5
7	2	4
Total	10	22

Table 2: Late assignments submitted per module

Mean Grade Point		
Module	Control	Experimental
1	71%(3.72)	75%(2.05)
2	79% (2.08)	71%(2.33)
3	73%(3.19)	73%(2.55)
4	62%(3.72)	66%(2.49)
5	74%(4.26)	75%(2.44)
6	67%(3.41)	67%(1.78)
7	56%(3.48)	65%(2.50)

Table 3: Mean grade points (with standard deviations) scored on the quiz by both groups

The data collected lays out some interesting points. The experimental group, at an anecdotal level, showed a greater inclination to submit the final assignment as compared to control group. Bear in mind that the experimental group students - by the time they submit the final assignment - have already submitted multiple assignments on the topic. A non-submission rate, that is almost half of the control group, may hint at the student's proclivity and willingness at submitting the final assignment. We believe that a better non-submission rate for the experimental group, even after doing multiple rounds of assignments is a healthy indicator of voluntary student engagement with the course.

Even though the non-submission rate is lower in the experimental group, the late submission rate is higher by over 100%. Late submissions in both group were allowed to see that if given the time, would students be motivated enough to work on the assignments? We found that students were more willing to work on the assignments in the experimental group even if that meant submitting it late. This is evident from the fact that there are more late submissions in experimental group than no submissions. The trend is reverse in the control group. This is to reiterate that the data presented here for experimental group is for the last cumulative assignment. By this time, for the same module, students would have submitted many incrementally difficult assignments, and a general student fatigue is expected which may speak for the higher number of late submissions.

Table 4 presents the end of module quiz grades for both groups. The groups were administered the exact same quizzes. There seems to no significant difference in the quiz performance for the groups, though the standard deviation in the experimental group seems to be on the lower side than that of the control group. Does that mean that constant practice, even though unable to improve overall group performance on quizzes, can help stem high variability of individual performance in the group? Could it be because weak students were able to improve their performance gradually? We cannot say anything for sure given such small sample size but the data does provide directions for potential explorations.

The groups were administered the exact same final exam. The two part exam consisted of writing a JAVA program and a multiple choice quiz that covered all seven modules. The JAVA program was worth two-third of the total points, and the quiz, one-third. Table 5 illustrates the data.

Group	Average Final Quiz Score	Average JAVA Program Score	Cumulative Average
Control	66%	51%	56%
Experi-mental	74%	71%	72%

Table 4: Final exam score for both groups

It is quite interesting to note that while there was no significant difference between module quiz scores, the experimental group performed much better in the final exam. Even though the gains in the final quiz are marginal, the experimental group outperformed the control group by 20% in JAVA program writing. The overall cumulative improvement in final exam mean score was 16%. These numbers may insinuate that−for the experimental group−the increased practice led to an improvement in final exam score, though it is too early to say anything with high degree of confidence due to such a small sample size. Nevertheless, the final exam numbers are encouraging.

No Submissions								
Module	1	2	3	4	5	6	7	Total
1	0	0	0	0	-	-	-	0
2	0	0	0	0	0	0	-	0
3	0	0	0	0	-	-	-	0
4	0	1	0	1	0	0	-	2
5	0	0	0	0	1	-	-	1
6	0	0	1	0	1	1	3	6
7	0	2	1	1	3	-	-	7

Table 5: Assignments not submitted

Intra Group Analyses

Table 6 and 7 present detailed assignment submission data for the experimental group. The first column represents the module/chapter that was covered, and the numbered columns represent the assignment number in that particular module. Some modules had four, some five, and some had seven assignments. The instances of no submissions are relatively very low as compared to late submissions. Similar trend was missing in the control group.

Late Submissions								
Module	1	2	3	4	5	6	7	Total
1	0	1	2	1	-	-	-	4
2	2	1	2	2	0	2	-	9
3	0	0	1	3	-	-	-	4
4	2	1	3	2	1	2	-	11
5	2	2	3	4	5	-	-	16
6	2	1	4	4	2	1	5	19
7	2	5	6	5	4	-	-	22

Table 6: Assignments submitted late

Table 8 presents a cumulative summary of the assignments. Cumulatively, only about 2% of the total assignments were not submitted. This could mean many things; one of the possible explanations might be that given the right conditions, the students were willing to engage more. Late submissions were allowed with reduced credit, and cumulative late submission rate stands at about 10.5%.

The instances of both late and no submissions increase as the course progresses, even though the rate of increase of no submissions is low as compared to late submissions. This may be explained by the fact that the concepts to be learned become complex as the course progresses, and some students might have given up on some of the assignments.

Module No	Maximum Possible Sub-missions	Not Sub-mitted	Late Sub-missions
1	88	0	4
2	132	0	9
3	88	0	4
4	132	2	11
5	110	1	16
6	154	6	19
7	110	7	22
Total	814	16(1.9%)	85(10.5%)

Table 7: Assignment Summary

End of Course Survey

With the experimental group, we also conducted an end of the course survey to gauge how AAAD was received by our students. Participation was 100%. The questions were primarily centered around the potential impact of high number of assignments on their motivation, stress levels, and their choice between AAAD and the usual method of single assignment per module used at our department. The full survey is listed in appendix C. A few questions are discussed in the following paragraph.

Effectiveness of AAAD

One of the questions asked the students about how they felt about the utility and effectiveness of AAAD in completing the course satisfactorily. A surprising 90% of the students answered that they felt positive/better about using this technique while 10% reported in negative, and answered that they felt slightly worse.

Another question asked the students about the utility of doing a daily assignment in learning computer programming. A whopping 100% of the students felt that it is useful. This gives us some confidence that given the right cognitive load and environment, students do see potential value in constant practice for learning programming.

AAAD vs Normal Course Delivery

Another important question asked the students about their choice between AAAD and the normal course delivery mechanism of doing one assignment per week. 95% of the students said that they would prefer AAAD. Hence, the students overwhelmingly choose AAAD as a mode of course delivery over our normal delivery method. This, we believe, is a very important piece of feedback for us.

Impact of AAAD on Student Stress Levels

Another very important question on the survey asked the students about their perception of stress levels about doing so many assignments. Half of the students answered that AAAD made it easy for them to manage stress, 32% said it increased their stress levels, and 18% choose that it made no difference. We were initially concerned that a high percentage of students might report increased stress levels. Just 18% students choosing higher stress levels came as quite a surprise. If this indeed is the case, it is one of the big incentives for us to continue to utilize, and improve this technique further.

4. DISCUSSION

With such a small sample size, it is quite early to generalize the utility of this technique, but the initial results do reveal some interesting insights. Most of the students seem to find AAAD beneficial, even if it means spending more time than usual to work on so many assignments.

Potential Strengths

According to the assignment data collected and student responses on the survey, it is clear that most students show an inclination towards practicing more as long as the cognitive load is manageable. This becomes clear from the minimal no-submission and late-submission instances during module 1 to 5 that cover basic JAVA concepts. Module 6 and 7 cover complex concepts such as 2D arrays and file operations. The instances of no-submission and late-submission rise during these modules. For future research, we contemplate breaking down the assignments further in module 5 and 6, to see if that would reduce the instances of late and no submissions. Overall, this technique, appears to successfully increase student engagement in the course.

Another strength is the high degree of acceptance students showed towards this technique. It seems that students engaged in the course not just because they were pushed by daily deadlines; they seemed to have embraced the method, and found value in it. Even if they had to spend more time consistently doing assignments, they argued that it helped them learn programming, and positively pushed them to engage themselves with the course.

Potential Limitations

It is no doubt that the workload of this technique may be perceived as higher when compared to orthodox course delivery. The pressure of completing an assignment every day can still lead to student frustration, and may even exacerbate the very factor the technique was designed to mitigate. Results and responses, however, show that the technique successfully navigated these roadblocks. It remains to be seen, if these results can be replicated in future courses.

Another significant potential limitation of this technique is its resource intensiveness. Since students have do so many assignments, they tend to ask many more questions about the concepts, as well as clarifications on assignments. Providing timely feedback is challenging even when the instructor has a course grader. Grading so many assignments, in our experience, was one of the major concerns, as this may inadvertently lead to grading fatigue. Future research will investigate simulated software and automatic grading systems to reduce this grading workload.

Another important aspect of employing this technique was the continual and immediate presence of instructor and tutor support. Without this perennial support, this technique may be rendered less effective. Our experience in a more traditional approach is that about 50%-60% of the class asked questions on assignments on the day the assignments were due. Since students have a due date almost every day of the week, AAAD requires continuous tutor support due to sheer volume of the queries. If these questions remain unaddressed at the outset, it may cause learning gaps for the students. Since the subsequent assignments build on previous assignments, it may have a snowball effect, which is highly undesirable.

Another very important point of concern is that many of our students work full time. For them, as evidenced by comments in the survey, it is difficult to schedule time every day to finish the assignments. The peculiar observation, however, is that even the full time working students appreciated AAAD technique; it is just that they find it difficult to schedule assignment time every day.

5. CONCLUSION AND FUTURE WORK

Students in our introductory programming course agree that an assignment a day technique added value to their process of learning computer programming.
AAAD helped them practice consistently, thereby improving their enthusiasm about the course. Though there was no significant differences in

the individual chapter quiz scores between the groups, the experimental group performed much better in the final exam.

Even though the students reported that they spent more time on the assignments, and had mixed reactions towards it, they overwhelmingly appreciated the value it brought to the table, and were convinced of its efficacy. The survey responses indicate that though the technique was very well received, it was not without its challenges. Firstly, grading a large number of assignments, and providing high volume of feedback is resource intensive. Continuous tutor support is also required to help stem student frustration, and to give them the feeling that help is always available.

Our future work includes finding ways to mitigate the load on the instructor, tutor/grader and students while maintaining the integrity of the technique, which is, continual practice and feedback. One aspect is the use of automatic grading systems to reduce the grading load. We also envisage coupling an automatic grading system with an artificial tutor bot capable of answering basic questions about the course, assignments, and programming simple concepts. Finally, we want to review the structure and design of the assignments to determine if there is a way to minimize questions. We are encouraged with this initial study and the promise of future research. We are contemplating using the same technique in our online programming course to see the technique's applicability in an online environment.

6. REFERENCES

Alexandron, G., Armoni, M., Gordon, M. & Harel, D. (2014). Scenario−based programming: Reducing the cognitive load, fostering abstract thinking. In Companion Proceedings of the 36th International Conference on Software Engineering (pp. 311−320). New York, NY: ACM.

Allan, V. H. & Kolesar, M. V. (1997). Teaching computer science: a problem solving approach that works. ACM SIGCUE Outlook, 25(1−2), 2−10.

Antonio Crego, María Carrillo-Diaz, Jason M. Armfield and Martín Romero Stress and Academic Performance in Dental Students: The Role of Coping Strategies and Examination-Related Self-Efficacy Journal of Dental Education February 2016, 80 (2) 165-172.

Beaubouef, T. B. & J. Mason (2005). Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. Inroads − The SIGCSE Bulletin, 37(2), 103−106.

Bennedsen, J. & Caspersen, M. E. (2007). Failure rates in introductory programming. ACM SIGCSE Bulletin, 39(2), 32−36.

Brabrand, C. & Dahl, B. (2009). Using the SOLO Taxonomy to Analyze Competence Progression of University Science Curricula. Higher Education, 58(4), 531−549.

Brown, S. W., & Bennett, E. D. (2002). The role of practice and automaticity in temporal and nontemporal dual-task performance. Psychological Research, 66, 80−89.

Eckerdal, A. (2009). Novice Programming Students' Learning of Concepts and Practice. Doctoral dissertation, Acta Universitatis Upsaliensis.

Glover, J.A., Ronning, R.R. and Bruning, R.H.: 1990, Cognitive Psychology for Teachers, Macmillan, New York.

Howles, T. (2009). A study of attrition and the use of student learning communities in the computer science introductory programming sequence. Computer Science Education, 19(1), 1−13.

Kalchman, M., Moss, J., & Case, R. (2001). Psychological models for the development of mathematical understanding: Rational numbers and functions. In S. M. Carver & D. Klahr (Eds.), Cognition and instruction: Twenty-five years of progress (pp. 1-38). Mahwah, NJ, US: Lawrence Erlbaum Associates Publishers.

Kaplan, D. S., Liu, R. X., & Kaplan, H. B (2005). School related stress in early adolescence and academic performance three years later: The conditional influence of self-expectations. Social Psychology of Education, 8, 3-17.

Kim, J. & Lerch, F. J. (1997). Why is programming (sometimes) so difficult? Programming as scientific discovery in multiple problem spaces. Information Systems Research, 8(1), 25−50.

Kinnunen, P. & Malmi, L. (2006). Why students drop out CS1 course?. In Proceedings of the Second International Workshop on Computing Education Research (pp. 97–108). New York, NY: ACM.

Lahtinen, E., Ala–Mutka, K. & Järvinen, H. M. (2005). A study of the difficulties of novice programmers. ACM SIGCSE Bulletin, 37(3), 14–18).

Luxton-Reilly, A. (2016). Learning to program is easy. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 284–289). New York, NY: ACM.

Margulieux, L. E., Catrambone, R. & Schaeffer, L. M. (2018). Varying effects of subgoal labeled expository text in programming, chemistry, and statistics. Instructional Science, 10.1007/s11251-018-9451-7.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I. &, Wilusz, T. (2001). A Multi–national, Multi–institutional Study of Assessment of Programming Skills of First–year CS Students. ACM SIGCSE Bulletin, 33, 125–180.

Mendes, A. J., Paquete, L., Cardoso, A. & Gomes, A. (2012). Increasing student commitment in introductory programming learning. In Frontiers in Education Conference (FIE) (pp. 1–6). New York, NY: IEEE.

Newman, R., Gatward, R. & Poppleton, M. (1970). Paradigms for teaching computer programming in higher education. WIT Transactions on Information and Communication Technologies, 7, 299–305.

Oliver, D., Dobele, T., Greber, M. &Roberts, T. (2004). This Course has a Bloom Rating of 3.9. In Proceedings of the Sixth Australasian Conference on Computing Education (ACE '04) (pp. 227–231). Darlinghurst, Australia: Australian Computer Society.

Rist, R. S. (1986). Plans in Programming: Definition, Demonstration, and Development. In Soloway, E. & Iyengar, S., eds., Empirical Studies of Programmers. Norwood, NJ: Ablex Publishing, pp. 28–47.

Rist, R. S. (1989). Schema Creation in Programming. Cognitive Science, 13, 389–414.

Rist, R. S. (1995). Program Structure and Design. Cognitive Science, 19, 507–562.

Rist, R. S. (2004). Learning to Program: Schema Creation, Application, and Evaluation. In Fincher, S. & Petre, M., eds., Computer Science Education Research. London, UK: Taylor & Francis, pp. 175–195.

Robins, A. V., Rountree, J. & Rountree, N. (2003). Learning and teaching programming: A review and discussion. Computer Science Education, 13(2), 137–172.

Rogalski J. & Samurçay R. (1990). Acquisition of programming knowledge and skills. In J. M. Hoc, T. R. G. Green, R. Samurçay & D. J. Gillmore, eds., Psychology of Programming. London: Academic Press, pp. 157–174.

Sheard, J. & Hagan, D. (1998). Our failing students: a study of a repeat group. ACM SIGCSE Bulletin, 30(3), 223–227.

Sorva, J. (2013). Notional machines and introductory programming education. ACM Transactions on Computing Education (TOCE), 13(2), Article 8 (31 pages).

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. Cognitive Science, 12(2), 257–285.

Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. Learning and Instruction, 4(4), 295–312.

Utting, I., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., Paterson, J., Caspersen, M., Kolikant, Y., Sorva, J. & Wilusz, T. (2013). A fresh look at novice programmers' performance and their teachers' expectations. In Proceedings of the ITICSE Working Group Reports Conference on Innovation and Technology in Computer Science Education (pp. 15–32). New York, NY: ACM.

Watson, C. & Li, F. W. (2014). Failure rates in introductory programming revisited. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (pp. 39–44). New York, NY: ACM.

Winslow L E (1996) Programming pedagogy – A psychological overview. ACM SIGCSE Bulletin, 28(3), 17–22.

APPENDIX A

Table 8: Increment in congnitive load with time

Assignment No.	Description	Concepts Tested	Cognitive Load
1	Write a method printS that takes a string as an input and prints it to the console.	Rudimentary method writing.	Low
2	Modify the above method printS and enable it to take another argument, an integer, *n*. The method then prints the string *n* times in a line.	Method writing, method calling, method modification.	Low
3	Reuse printS to print a user entered string $n \times n$ *times* i.e a square with each element as the string	User input, loops, method writing, method calling	Medium
4	Reuse printS method to print a right angle triangle in terms of user entered string	User input, loops, method writing, method calling, Problem solving	Medium
5	Reuse printS to print a pyramid in terms of user entered string	User input, loops, method writing, method calling, Problem solving	High

APPENDIX B

Artifact: *Assignment_5_1*

Write a static method called *printS* that is passed a String *s* as an argument. The method prints the passed string to the console and returns nothing. Write a `main` method that allows the user to enter the string from the keyboard. Write error-checking code wherever possible.

Artifact: *Assignment_5_2*

Modify the *printS* method written in Assignment_5_1 to enable it to include another argument of type integer *n*. The method then prints the passed String *s* to the console a total of *n* times. Write a `main` method that allows the user to enter the string and the integer from the keyboard. Write error-checking code wherever possible.

Artifact: *Assignment_5_3*

Reuse the *printS* method written in Assignment_5_2 to enable it to print a *n*×*n* matrix of the passed String *s*. Write a `main` method that allows the user to enter the String and the integer from the keyboard. Write error-checking code wherever possible.

Artifact: *Assignment_5_4*

Reuse *printS* method that you wrote in Assignment_5_3, to write a method called *printTriangle* that is passed two arguments, an int `n` and a String `s`. It should print a right triangle in which the base of the triangle is made of `n` copies of `s`, and the vertex of the triangle has a single copy of `s` on the right. For example, calling *printTriangle* `(13, "*");` prints the following lines:

```
            *
           **
          ***
         ****
        *****
       ******
      *******
     ********
    *********
   **********
  ***********
 ************
*************
```

You will call *printS* from within *printTriangle.* Write a `main` method that calls *printTriangle* `(13, "*")`.

Some parts adapted from Big Java Late Objects by Cay S. Horstman

Artifact: *Assignment_5_5*

Write a method called *printPyramid* that is passed an odd integer `n` and a String `s`, and that prints a pyramidal shape using `s`. The top of the pyramid has a single copy of `s`, and each successive row has two additional copies of `s`. The last row contains `n` copies of `s`. You must reuse *printS* method written in previous assignments to accomplish this task. For example, calling *printPyramid* `(21, "*");` prints the following lines:

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
 *******************
*********************
```

Test your work by calling *printPyramid* `(21, "*")` from the `main` method.

Some parts adapted from Big Java Late Objects by Cay S. Horstman

APPENDIX C

CSE 174: Student experiences with multiple assignments

English ▼

SURVEY INSTRUCTIONS

Dear CSE 174 Student,
This short survey is designed to ask you about your experiences in this course, specifically about an assignment a day (AAAD) format, where, for each chapter, you did one assignment per day (or more) depending upon the difficulty level of the assignment(s). Please consider each question carefully. Your participation is much appreciated.

Student Resources

Did the daily assignments prepare you for the last (concluding) assignment of the module?

Definitely yes Probably yes May be Probably not Definitely not

Did the daily assignments prepare you for the midterm and final exams?

Definitely yes Probably yes May be Probably not Definitely not

How difficult was it for you to ***schedule time*** every day to complete the daily programming assignment?

Extremely easy | Moderately easy | Slightly easy | Neither easy nor difficult | Slightly difficult | Moderately difficult | Extremely difficult

How difficult was it for you to ***complete*** the daily assignment?

Extremely easy | Moderately easy | Slightly easy | Neither easy nor difficult | Slightly difficult | Moderately difficult | Extremely difficult

Overall, how much time did you spend on completing the daily assignment?

A great deal	A lot	A moderate amount	A little	None at all

How did the daily assignment make you feel about your ability to complete the course satisfactorily?

Much better	Moderately better	Slightly better	About the same	Slightly worse	Moderately worse	Much worse

Overall, how useful is a daily assignment for learning computer programming?

Extremely useful	Moderately useful	Slightly useful	Neither useful nor useless	Slightly useless	Moderately useless	Extremely useless

Given an option, what mode of practice work would you prefer for this course?

One long and possibly difficult assignment each week

One small and possibly easy to medium difficulty assignment every day that builds on previous concepts

No preference

Block 2

How did doing multiple assignments effect your stress levels?

It made it easy to manage overall stress as the assignments were gradually increasing in difficulty

It increased my stress as I had to do many assignments

It made no difference

Did having a programming assignment everyday format encourage you to practice more on your own?

It positively pushed me to practice much better	It made me practice moderately better	It made me practice slightly better	I would have practiced a lot regardless of this format	It made me practice less

	Extremely well	Very well	Moderately well	Slightly well	Not well at all
Outcome 1: Use and describe a contemporary programming language and programming environment (IDE) like Dr. Java.	○	○	○	○	○
Outcome 2: Identify and eliminate errors in programs	○	○	○	○	○
Outcome 3: Specify, trace, and implement programs written in a contemporary programming language like Java that solve a stated problem in a clean and robust fashion	○	○	○	○	○
Outcome 4: Solve programming problems using a procedural approach i.e. divide your program into methods	○	○	○	○	○
Outcome 5: Describe, trace, and implement basic algorithms like linear search, binary search etc.	○	○	○	○	○
Outcome 6: Apply and communicate information that they read from technical sources such as APIs like Scanner etc.	○	○	○	○	○