An Investigation on Student Perceptions of Self-Regulated Learning in an Introductory Computer Programming Course.

Pratibha Menon
menon@calu.edu

Department of Computer Science,
Information Systems and Electrical Technology
California University of Pennsylvania
California, PA, 15419

Abstract

Learning how to become a self-regulated learner could benefit students in introductory undergraduate courses, such as computer programming. This study explores the perceived value of instructional and skill-building activities and students' self-efficacy to learn and apply programming skills in an introductory computer programming course. The instructional activities include code-demos through which the instructor demonstrates several cognitive strategies for self-regulated learning. Four different skill-building activities accompanied by Q&A sessions let the students model the teacher's practices, and apply various self-regulated learning methods to strengthen their programming skills. Surveys are implemented and analyzed to learn the students' perceptions of the task value of skill-building activities and the Q&A sessions and their reported self-efficacy for independent mastery, problem-solving, correcting errors, and experimenting with programs. Studies revealed that the perceived ability to master programming independently is significantly correlated to the perceived task value of activities that required students to complete programs similar to the instructor's code-demos. Students who report a higher self-efficacy for problem-solving also positively value the Q&A sessions through which they obtain help from the instructor to complete tasks on pre-written codes.

Keywords: Computer-programming, Self-Regulated-Learning, self-efficacy, problem-solving, teaching, learning.

1. INTRODUCTION

Introductory computing courses are generally regarded as difficult and often see many dropouts that lead to attrition (Kinnunen & Malmi, 2006). According to (Beaubouef & Mason, 2005), most attrition occurs during freshman and sophomore years. Studies have also shown that students often do not acquire good practice as they complete their introductory computing courses (Lister et al., 2004). One approach to increasing success rates in undergraduate computer programming courses is teaching students how to become more effective self-regulated learners who will apply deliberate practice to improve their programming skills.

Self-regulated learning (SRL) is an active process in which the learners take significant initiative in their learning process and persevere by continually adapting to the tasks at hand (Zimmerman, 2002). They set learning goals, monitor their goals, and regulate their cognition, motivation, and behavior to achieve their set goals (Pintrich, 2004).

A key determinant of whether learners employ SRL depends on the beliefs about their capabilities (Cleary& Zimmerman, 2006). SRL

strategies and beliefs of self-efficacy to do a particular task are interdependent; both require the presence of specific cognitive capacities, such as the ability to set goals, self-monitor, reflect, and make judgments. More importantly, both also support personal agency or control. Students' academic self-efficacy is related to motivation and academic achievement (Komarraju & Nadler, 2013). Self-efficacy combines judgments of one's ability to accomplish a task, confidence in one's skills to perform a task, and expectancy for success in the task.

Another motivational factor that positively impacts SRL is the learner's task value (Pintrich, 2000) (Pintrich & Zusho, 2002). A student can perceive a task as valuable based on its perceived utility, importance, or interest.

Improving students' self-efficacy in learning through SRL strategies could significantly benefit students in an introductory computer programming course (Bergin et.al, 2005). The majority of learning in a computer programming course occurs outside the classroom, as it involves hands-on practice in problem-solving, writing, compiling, and testing computer programs. However, many college students do not know how to effectively self-regulate their learning process (Bembenutty, 2008). First-year students often rely on their teachers' support during secondary schooling to direct their learning processes (Chemers et.al, 2001). Therefore, many freshmen students find it challenging to engage in self-directed learning that requires repetitions of planning-practice-and-reflection cycles.

This study explores the value of various instructional and skill-building activities that can teach some of the critical self-regulated learning strategies required to master computer programming. This study's context is an undergraduate level introductory programming class of 22 students in a computer information system program at a public university. This study considers an instructional approach that models the instructor's practice of applying the SRL process during a programming demonstration and Q&A sessions. Additionally, this study also considers four different skill-building activities through which students get an opportunity to emulate the instructor's practices to solve programming problems and develop critical skills.

This study attempts to find the correlation between the student perceptions of the learning activities' task value and the reported self-efficacy for independent mastery, problem-solving, correcting errors, and experimenting with programs.

2. RELATED WORK

This study assumes that learning computer programming practice occurs as a cyclical exchange of knowledge and information between the learner and an external learning environment. Besides the learner's interaction with external agents, a learner goes through an internal process that regulates the thoughts and actions within the learner's mind. A Self-Regulated-Learning (SRL) model is used to identify various steps in a learning process.

2.1 The teaching-learning model

For this study, the learner's interaction with the learning environment is assumed to occur in two ways; 1) between the learner and the teacher, and 2) between the learner and an external learning tools such as an Integrated Development Environment (IDE). These interactions may be termed as the Teacher-Practice cycle and the Teacher-Modeling cycle, respectively (Laullilard, 2012). The Teacher-Practice cycles involve interactions in which the teacher demonstrates the ideal way to practice a skill and provides useful feedback to the students to improve their skill. On the other hand, the teacher-modeling cycle allows the student to independently model the teacher practices and involves an interaction between the learner and the learning tool, which in this study is the IDE. Teacher-Modeling cycles influence the learner's abilities to engage in independent and deliberate practice to improve programming and problem-solving skills. The IDE provides immediate feedback to students and provides opportunities for students to engage in repeated practice and self-regulated learning. Although there might be several relevant interactions among the learners, which are beyond this paper's scope.

In a programming course, the Teacher-Practice cycle typically consists of code-demonstrations and Q&A sessions used to discuss coding and problem-solving practices. The Teacher-Modeling cycle is enabled through skill-building problems that require the use of an IDE to implement solutions. A teacher may provide additional feedback and support through regular Q&A sessions to help students understand and apply appropriate actions based on the IDE feedback.

2.2 The Self-Regulated Learning Model

Self-Regulated-Learning (SRL) is a research area under which many variables that influence learning, such as self-efficacy, volition, and

cognitive strategies, are studied within a comprehensive and holistic approach. A meta-analytic study of SRL identifies various models that researchers can utilize to suit their research goals better and focus (Panadero, 2017). This study draws from previous studies on SRL that posits that Self-regulated learning can be taught (Pintrich & Zusho, 2002). SRL strategies can be transferred to students through instructions specific to the learning context (Perels, Dignath, & Schmitz, 2009). These studies show that providing direct instructions on specific strategies and using the right learning environment can enhance students' self-regulated learning.

This study's SRL model is derived from Zimmerman's work (Zimmerman & Moylan, 2009). Zimmerman's SRL model is organized into three phases: forethought, performance, and self-reflection. In the forethought phase, the students analyze the task, set goals, and plan how to reach them.

Students execute the task in the performance phase as they monitor their progress and use self-control strategies to keep themselves cognitively engaged and motivated to finish the task. Finally, in the self-reflection phase, students assess and understand the factors that might have impacted their success or failure. The self-reflection phase generates reactions that can positively or negatively influence how the students approach the task in later performances. Zimmerman's cyclical phase model has been tested in a series of studies. Studies that compare experts and non-experts in sports show that experts performed more SRL actions (Cleary & Zimmerman, 2001) (Cleary et al., 2006).

Zimmerman's three-phase SRL model could be applied to model the learning process in a computer programming course. Students need to analyze the task requirements in a programming course and continuously monitor their code to find errors before arriving at an acceptable programming solution. After completing a task, it would help the students reflect on their coding habits and practices to improve their performance. By providing students with suitable instruction during the Teacher-Practice cycles, the teacher can model different ways by which students may monitor their practices. Students could apply these learning strategies to take control of their learning during the Teacher-Modeling cycles.

Previous studies have examined the role of self-regulation within the educational context of computer programming (Bergin et al., 2002) (Kumar et al., 2005) (Chen, 2020) (Ramirez et al., 2018).

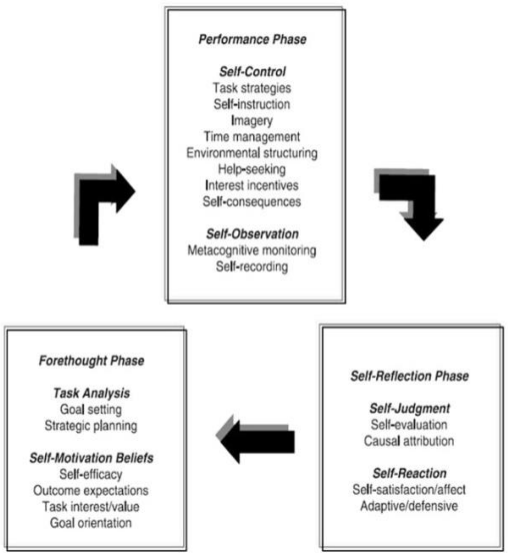

Figure 1. Zimmerman's Self-Regulated-Model (Zimmerman & Moylan, 2009)

These studies' focus has been to evaluate the impact of self-regulated learning strategies on students' coding performance. Another study by Castellanos et al. uses students' source code to study student motivation, performance, and learning strategies (Castellanos, 2017). Unlike the two studies mentioned earlier, the study described in this paper evaluates the perceived value of teaching and learning activities and the self-efficacy required to learn and practice programming through a course that instructs cognitive strategies for self-regulated learning through the course contents. This study focuses on students' self-efficacy and not on their measured or reported performance in the course.

Extensive recent work on SRL exists in building online, adaptive learning systems with open-learner-models (OLMs) that allow learners to visualize and inspect their progress during the learning process. It has been pointed out that OLM can support metacognition and self-regulation (Bull & Kay, 2013). Moreover, researchers have incorporated OLM in all phases of self-regulation, i.e., preparation, performance, and appraisal, and in the areas of cognitive, metacognitive, motivational, and emotional support (Hooshyar et al., 2020). For example, OLM has been used to improve self-assessment

accuracy through dialog-based support (Suleman et al., 2016) and improve engagement in a programming course (Hossieni et al., 2020). All these studies were performed in full-online learning that does not include any teacher's direct intervention during the learning process. The study described in this paper models a typical freshman-level, under-graduate classroom scenario. The teacher still plays a central role in mediating students' self-regulation strategies. Therefore, this paper's focus is on a teaching-learning model that includes the central role of a teacher in designing and supporting the learning process by adapting to the learners' needs.

3. THE DESIGN APPROACH

The instructional design evaluated in this study incorporates teaching strategies for the Teacher-Practice cycle and suitable learning activities for the Teaching-Modeling cycle. The teaching methods are chosen such that they incorporate the three phases of Zimmerman's SRL model.

3.1 Designing the Instructional Activities

The teacher-practice learning cycle consists of activities through which the instructor, who is also an expert programmer, models the programming practices. Table 1 shows the instructional activities in the Teacher-Practice cycle. Code-demonstrations (code-demos) explain the programming process through task analysis, code development, execution, and testing. The sample code used for the code-demo contains extensive documentation and comments that students can refer to later on.

The forethought/planning phase of the code-demo typically includes a detailed explanation and analysis of the problem statement to identify the functional and data requirements. These planning activities are written down as part of the code documentation in the code's comments section. The instructor may use real-world examples to show the value of the problem. The instructor will then teach students to identify the problem's inputs and the expected outputs, create a test plan, and search for similar problems that use similar programming structures.

The code-demo's performance phase typically involves the instructor elaborating on the systematic thought process required to write the program sequences. The instructor encourages extensive use of comments next to the code statements. Students also observe how the instructor applies techniques such as tracing the variables or printing out the variables' values to test and incrementally build their code.

	Instructional activities - Teacher-Practce Learning Cycle		
	Forethought	**Performance**	**Self-Reflection**
Code Demos	Problem Analysis, Solution planning, Reviewing Test Plans	Choice of constructs, Identifying right sequence, Tracing variables, Running Tests	Evaluating Style & Practices and Errors
Q&A Sessions	Task planning , Goal Setting for the class	Discussions on Identifying and correcting errors; adopting good practices	Choosing practice materials to strengthen practice

Table 1. Instruction Activities – Teacher-Practice Learning Cycle

The self-reflection phase of each code-demo is used to analyze various options for accomplishing the same outputs. The instructors discusses acceptable coding practices that are relevant to the problem. The instructor also highlights the challenges commonly encountered while solving the problem and improve their problem solving and programming skills.

Integral to the Teacher-Practice learning cycle are the Q&A sessions. The Q&A sessions are conducted during the regular class session after students get adequate time to complete learning activities. These activities are described in Table 2. During the Q&A sessions, the instructor would clear any misconceptions or problem-solving difficulties students would have experienced while completing a learning activity. The instructor may also discuss the graded assignments and some of the common errors and misconceptions that would have appeared in student submissions.

3.2 Designing Practice Exercises

The Teacher-Modeling cycle follows the Teacher-Practice cycle. Students learn to apply the teacher's program development practices previously explained through the code-demos. Through shorter practice problems, such as the Test-Tube, Hack-the-code, Messed-up-code, students practice essential programming skills that could be used to develop larger programs.

The Do-It-Yourself (DIY) exercises are more time-consuming activities that students complete at home. These activities contain problems that are analogous to the ones explained during code-demos. By observing the sample code provided during the code-demos, students can recollect and emulate the practices of the instructor and apply all three phases of SRL to document and write the code by themselves using an IDE. The

DIY activities also advise students to analyze the problems and the test plan, write extensive comments, and incrementally build their code. A sample DIY activity problem is shown in Appendix B.

Activity name	Practice Exercises - Teacher-Modeling Learnng Cycle
DIY	Try out every code-demo independently, following the instructor's comments/explanation.
Test-Tube	Test a given code by varying the inputs,or by making suggested changes to obtain a given output
Messed Up Code	Analyze an errored-code
Hack the Code	Experiment with a given code to produce a set of outputs (including errors)

Table 2. Type of Practice Exercises – Teacher-Modeling Cycle

As they learn to write programs, novice programmers generate programming errors, and they need to learn how to identify the cause and correct these errors. Many students require help to understand the types of errors and on how to recall their previous troubleshooting experiences to improve their programming skills.

The instructor developed activities called Hack-the-code and Messed-up-code to help students gain practice and become comfortable with detecting and correcting logical, syntax and runtime errors. The Messed-up code contains one or more errors that students need to identify and correct. Hack-the-code is an activity in which students need to alter a pre-written code's logic to obtain the required set of outputs. The Messed-up code and Hack-the-code activities intend to encourage students to feel comfortable in experimenting with their code. Another activity that encourages students to solve problems by experimentation is the Test-Tube activity. This activity requires students to develop and execute a test-plan for a given code and, in most cases, also requires them to trace the variables. All these activities intend to teach cognitive and meta-cognitive strategies that improve coding practice. Appendix B contains Samples of Hack-the-code, Test-Tube, and Messed-up code activities.

The learning activities let students work on the problems by themselves and learn how to ask for help from their peers and instructor. Students are encouraged to apply the three phases: task analysis, performance monitoring, and self-reflection for every task they perform. The Q&A sessions address the problems students faced while working on the activities. Students attempt the smaller activities during class time, and the more extensive DIY activities are completed at home. Students received class participation points for attempting and not necessarily completing these activities. These activities prepare students to complete graded assignment problems and the exams.

4. THE STUDY

This study's primary intent is to analyze students' perceptions of the task value and their self-efficacy in an introductory programming course. This study is conducted in an undergraduate computer programming course that teaches introductory programming using Java. Results of a final, end-of-the-course survey are used to study the student perceptions of the usefulness of various learning activities and students' perceived self-efficacy to learn computer programming. Appendix A shows the final survey questions. An initial survey during the beginning of the course was also used to assess the learning needs of the incoming students. The questions of the initial survey are as listed in Table 3.

The surveys used a 5-point Likert scale to score student responses. Nineteen students attempted the final survey, and 20 students attempted the initial survey. Student surveys are administered anonymously during class time. Students were required to attempt all the assigned skill-building/learning activities assigned throughout the course. Practicing these skill-builder activities could potentially give enough cognitive and learning skills to help students regulate their efforts towards writing good computer programs.

5. RESULTS

5.1 The need for instruction of skills to learn to program

Table 3 shows the results of an initial survey conducted during the first week of the course. Students were less concerned about how much they could master this course's contents than about having the right skills and abilities to learn to program. This survey was administered to students during the first week of the course after the instructor discussed the course syllabus.

Table 3 indicates the self-reported prior experience with computer programming. Since data collected using a 5-point Likert scale is ordinal, a Spearman-rank correlation method is used to investigate the correlation between the degree of prior exposure to computer programming and students' learning concerns. Prior exposure to programming negatively correlates with a moderately significant correlation coefficient (rho of -0.6, $p = 0.005$)

with the students' concerns about having the right skills to learn to program.

	Very Much Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Very Much Agree
I am concerned about how much I can master the subject matter	2	3	2	7	6
I am concerned if I have the right skills to learn programming	1	6	6	3	4
I have some prior knowledge of programming	8	3	2	3	4

Table 3. Survey response distributions on the perceived self-efficacy to learn to program - before attending the course.

The correlation between prior exposure to programming and concerns about learning the subject matter was not significant (rho = -0.3, p=0.15). These results show that students with lesser prior exposure to programming were more concerned about having the right skills to learn to program than their concern about mastering the subject matter. These results pointed to the possibility that an instructional strategy that included explicit activities to build essential learning skills might be valuable to develop students' perceived self-efficacy in their ability to learn to program.

5.2 Perceptions of the value of learning activities in the course

A final survey administered at the end of the course showed the perceived value of various learning activities that became a regular part of instruction throughout the semester. Appendix A lists the final survey questions. Table 4 shows some of the results of the final survey. Most students agreed that practicing and participating in these learning activities were valuable in acquiring the programming skills that they were expected to learn from the course.

All the activities, except for the Q&A sessions, required students to apply their knowledge and skills to identify the problem, plan the solution, write the code, correct errors, and test the code –all by themselves. These activities provided students with different ways to apply one or more SRL skills related to learning how to develop programming solutions. The Q&A sessions were the time when students obtained help and feedback from the instructor.

	Very Much Disagree	Disagree	Neutral	Agree	Very Much Agree
Q&A sessions	0	0	1	11	7
Messed Up Code	0	0	1	12	6
Hack the Code	0	0	1	11	7
Test Tube	0	0	1	7	11
DIY	0	0	4	13	2

Table 4. Student response distribution on the effectiveness of different learning activities in developing programming skills

Survey results on students' perceptions, depicted in Table 5, showed that 18 out of 19 respondents agree or strongly agree that they feel comfortable experimenting with their code.

	Very Much Disagree	Disagree	Neutral	Agree	Very Much Agree
I feel that learning how to program has improved my problem solving skills	0	0	3	7	9
I feel confident to experiment with my programs	0	0	1	8	10
I feel confident that I can correct programming errors	0	0	0	9	10
I believe that one can master programming only by working on independently on hands-on activities	1	2	6	7	3

Table 5. Student response distribution on various indicators of student self-efficacy related to learning programming

All the respondents also report that they feel confident in their ability to correct programming errors. Out of the 19 respondents, 16 (85% of respondents) feel that learning how to program has improved their problem-solving skills. However, nearly 9 out of 19 (48%) respondents do not believe that they can master programming only by working independently on hands-on activities. At the same time, high value of the Q&A sessions, as shown in Table 4, shows that students have relied on getting help from the instructor through the Q&A sessions.

5.3 Correlation studies

A Spearman-Rho correlation was used to study the co-occurrence of various factors indicating self-efficacy (listed in Table 5) and the perceived value of various instructional methods (listed in Table 4). Table 6 shows the value of rho and p values after correlating the student responses. For the sample of 19 respondents, there existed a significant correlation (rho = 0.6, $p < 0.01$) between students' belief in their ability to independently master programming and the perceived value of doing many DIY activities.

	Task Value of Q&A	Task Value of DIY	Task Value of Messed-up-code	Task Value of Hack-the-code	Task Value of Test-Tube
Improved problem solving skills	**(0.57, 0.01)**	*(0.308, 0.13)*	*(0.07, .8)*	*(0.21, 0.38)*	*(0.21, 0.38)*
Experiment with programs	*(0.4, 0.08)*	*(0.25, 0.56)*	*(-0.06, 0.65)*	*(0.03, 0.9)*	*(0.05, .8)*
Correct programming errors	*(0.33, 0.16)*	*(0.18, 0.44)*	*(-0.16, 0.68)*	*(0.5, 0.5)*	*(0.5, 0.5)*
Master programming only by doing indepedent hands-on activities	*(0.4, 0.08)*	**(0.6, 0.006)**	*(0.46, 0.04)*	*(0.3, 0.20)*	*(0.19, .40)*

Table 6. Correlation results showing values of rho and p

Another significant correlation (rho = 0.6, $p < 0.01$) existed between the value of the Q&A sessions and the perception that learning to program has improved their problem-solving skills. No significant correlation was found to ascertain that the perceived values of Test-tube or Hack-the-code are associated with any of the factors that indicate the perceived self-efficacy measure listed in Table 5. A moderately strong correlation was seen between the value of messed-up code and the perceived ability to master programming independently.

5.4 The Instructor's reflection on the results

Both the Q&A session and the DIY activities involved the instructor's support to a much greater extent than the Test-tube, Hack-the-code, or the Messed-up-code activities. The DIY activities problems were very similar to those used in the code-demos to explain problem-solving. The code-demos provide a scaffold for students to work on their DIY problems. However, the DIY activities did require students to read the question prompt, discover a similar problem in the code-demos, write the solution, implement the code, debug, and test the codes with various inputs. The DIY activities resembled mini-projects, while the other learning activities were shorter problem-solving activities. Students were provided with a pre-written code for the Messed-up-code, Test-tube, and Hack-the-code activities. The value of completing the DIY programs by 'walking in the instructor's shoes' seems to correlate more with the belief that students can master programming through independent practice.

By reflecting upon the classroom experience, it was observed that students did not require much help from the instructor to complete the DIY activities. This could be due to the fact that the DIY closely resembled the examples in the code-demos that had extensive documentation corresponding to the planning, reflection and implementation phases of SRL. However, to complete the Test-tube, Hack-the-code, and Messed-up-code activities, students had no template to work with and had to recall similar problems or situations from their memory. As a result students required more help from the instructor for these activities. Majority of the Q&A sessions addressed ways to reformulate the task and identify similar problems from experience.

From an instructor's perspective, asking questions and seeking help is an important skill required to become independent, self-directed learners. A student who considers Test-tube and Hack-the-code as valuable to their learning is still not likely to say that they believe they can master programming independently, possibly because they needed more help and support to complete the tasks. Compared to the Test-tube and Hack-the-code activities, the Messed-up-code, which moderately correlated with belief in independent-mastery, did not require students to alter the inputs. A significant correlation between confidence in problem-solving skills and the value

of Q&A indicates that students are likely to view help and support as factors that improve their problem-solving, but not necessarily towards developing independent-mastery.

In addition to needing more help with the Test-tube, Messed-up-code, and Hack-the-code, students tended to make more mistakes, even though they would eventually figure out a way to correct the mistakes. From an instructor's perspective, learning how to correct mistakes indicates self-regulated learning. However, if students perceive mistakes negatively, they are less likely to register these activities as contributing to their confidence to learn independently. Despite their perceived task value, Test-tube, Messed-up-code, and Hack-the-code, they were not significantly correlated to confidence for independent mastery.

6. CONCLUSIONS

This study investigates the student perceptions of the role of teacher-practice activities and teacher-modeling activities in an introductory computer programming class. The majority of the students agree that all the hands-on learning activities had significantly helped them acquire the programming skills, even though more than half of the students reported that they were not confident in their ability to master programming independently. Emulating the instructor's coding process through the DIY activities is what the students found as most valuable in mastering their programming skills independently, and the Q&A sessions were strongly perceived and correlated with confidence in problem-solving skills. Future iterations of the course could consider tweaking the self-directed learning activities so that students can see the value of making mistakes and getting help as an essential part of their ability to master programming independently. Future studies could look into learning strategies that could help students regulate their behavior and motivation at a granular level as they encounter learning challenges.

7. ACKNOWLEDGEMENTS

The author would like to thank and acknowledge the PASSHE-FPDC grant for funding the instructor during summer 2019. The grant funding helped the author acquire the required professional development on Self-Regulated Learning and Design Thinking applied in this study.

8. REFERENCES

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin, 37*(2), 103-106.

Bembenutty, H. (2008). The teacher of teachers talks about learning to learn: An interview with Wilbert (Bill) J. McKeachie. Teaching of Psychology, 35, 363–372.

Bergin, S., Reilly, R., & Traynor, D. (2005) "Examining the role of self-regulated learning on introductory programming performance", Proceedings *of the first international workshop on Computing education research*, pp. 81-86, 2005.

Bull, S & Kay, J. (2013) "Open learner models as drivers for metacognitive processes," in International Handbook of Metacognition and Learning Technologies. Springer, pp. 349–365.

Castellanos, F. H. Restrepo-Calle, F., González. A, & Echeverry, J. R. (2017) "Understanding the relationships between self-regulated learning and students source code in a computer programming course," *2017 IEEE Frontiers in Education Conference (FIE)*, Indianapolis, IN, 2017, pp. 1-9,.

Chemers, M. M., Hu, L. T., Garcia, B. F. (2001). Academic self-efficacy and first year college student performance and adjustment. Journal of Educational psychology, 93(1), 55–64.

Chen, C. S., (2002) "Self-regulated learning strategies and achievement in an introduction to information systems course", *Information technology learning and performance journal*, vol. 20, no. no. 1, pp. 11.

Cleary, T., Zimmerman, B. J., & Keating, T. (2006). Training physical education students to self-regulate during basketball free throw practice. *Res. Q. Exerc. Sport* 77, 251–262.

Cleary, T., & Zimmerman, B. J. (2012). "A cyclical self-regulatory account of student engagement: theoretical foundations and applications," in *Handbook of Research on Student Engagement*, eds S. L. Christenson and W. Reschley (New York, NY: Springer Science), 237–257.

Hooshyar, D. M., Pedaste, K., Saks, ¨A., Leijen, E. Bardone, & Wang, M. (2020) "Open learner models in supporting self-regulated learning in higher education: A systematic

literature review, "Computers & Education, p.103878

Hosseini, R., Akhuseyinoglu, K., Brusilovsky, P., Malmi, L., Pollari-Malmi, K., Schunn, C., and Sirkiä, T. (2020) Improving Engagement in Program Construction Examples for Learning Python Programming. *International Journal of Artificial Intelligence in Education*.

Kinnunen, P., & Malmi, L. (2006). *Why students drop out CS1 course?* Paper presented at the Proceedings of the second international workshop on Computing education research.

Komarraju, M. & Nadler, D. (2013). Self-efficacy and academic achievement: Why do implicit beliefs, goals, and effort regulation matter? Learning and Individual Differences, 25, 67–72.

Kumar, V., Winne, P., Hadwin, A., Nesbit, J. et al., (2005) "Effects of self-regulated learning in programming", *Advanced Learning Technologies. ICALT 2005. Fifth IEEE International Conference on*, pp. 383-387.

Laurillard, D. (2012) *Teaching as a Design Science*. (New York: Routledge).

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *ACM SIGCSE Bulletin*, *36*(4), 119-150.

Panadero E. (2017). A Review of Self-regulated Learning: Six Models and Four Directions for Research. *Frontiers in psychology*, *8*, 422.

Perels, F., Dignath, C., & Schmitz, B. (2009). Is it possible to improve mathematical achievement by means of self-regulation strategies? Evaluation of an intervention in regular math classes. *European Journal of Psychology of Education, 24*(1), 17.

Pintrich, P. (2004). A conceptual framework for assessing motivation and self–regulated learning in college students. Educational Psychology Review, 16, 385–407.

Pintrich, P. R. (2000). The role of goal orientation in self-regulated learning. In M. Boekaerts, P.R. Pintrich, & M. Zeidner (Eds.), *Handbook of self-regulation (pp. 451–502)* (pp. 451–502). San Diego, CA: Academic Press.

Pintrich, P. R., & Zusho, A. (2002). *The development of academic self-regulation: The role of cognitive and motivational factors.* In A. Wigfield & J. S. Eccles (Eds.), *A Vol. in the educational psychology series. Development of achievement motivation* (p. 249–284). Academic Press.

Ramírez, J. J. E., Rosales-Castro, L. F., Restrepo-Calle & González, F. A. (2018) "Self-Regulated Learning in a Computer Programming Course," in IEEE Revista Iberoamericana de Tecnologias del Aprendizaje, vol. 13, no. 2, pp. 75-83.

Suleman, R. M., Mizoguchi, R. & Ikeda, M. (2016) "A new perspective of negotiation-based dialog to enhance metacognitive skills in the context of open learner models," International Journal of Artificial Intelligence in Education, vol. 26, no. 4, pp. 1069–1115, publisher: Springer.

Zimmerman, B.J. (2002). Becoming a self-regulated learner: An overview. *Theory Into Practice, 41* (2), 64-70.

Zimmerman, B. J., & Moylan, A. R. (2009). Self-regulation: Where metacognition and motivation intersect. In *Handbook of metacognition in education* (pp. 311-328). Routledge.

Appendix A

Final Survey - conducted at the end of the course					
Please answer these questions based on your learning experience in the CIS 120 course	**Very Much Disagree- 0**	**Disagree- 1**	**Neutral- 2**	**Agree - 3**	**Very Much Agree-4**
The Q&A session is a valuable learning method for this course					
Test-Tube: Experiementing with code is a valuable learning method for this course					
DIY : Trying out the code-demos using Eclipse is a valuable learning method for this course					
Messed-up-code: Analyzing and fixing an errored code is a vauable learning method for this course					
Hack-the-code: Experimenting with the code to alter the outputs helped me learn better					
I believe that one can master programming by working only independently on hands on activities.					
I feel that learning how to program has improved my problem solving skills					
I feel confident to experiment with my programs					
I feel confident that I can correct programming errors					

Appendix B

1. A Sample DIY problem:

Shopping Cart – Create a file called ShoppingCart.java

Please refer to the code demo called **VariableDataEntry.java** prior to attempting this problem. This problem shows you how to:

- obtain data from the user, scan this data, and save it in an appropriate variable.
- perform arithmetic using the numeric data types,
- print a message displaying values of all the variables.

In this program you will capture data of an item for a ShoppingCart application. Your program may need to know the following properties: customer_name, item_name, item price, sales tax rate, item quantity, calculated total price of all items in the cart

A ShoppingCart may need the following behaviors:

- Obtain the following data from the user for a single item: customer_name, item_price, sales_tax_rate, item_quantity. Scan these values and store them in variables of appropriate data type.
- Calculate the total price of all items in the cart
- Print a message listing all the item variables with its total calculated price (that includes the sales_tax factored in).

2. A Sample Hack-the-Code activity:

Refer to the code called AgeCheckerCase2.java.

```
import java.util.Scanner;
public class AgeCheckerCase2 {
    public static void main(String[] args){

        //Declare the input variable
        int age = 0;

        //Declare output variable
        double ticketPrice = 0.0;

        //Declare other variables
        double salesTax = 0.05;

        //Decision structure
        if(age < 12) {
            salesTax = 0.2;
        }

        ticketPrice = ticketPrice*(1+salesTax);

    }
}
```

Hack this code so that your decision structure calculates the ticketPrice based on the following rule: For an age that is less than 12, give a 20% discount on ticketPrice, but for an age greater than 65, give just 10% discount on the ticketPrice for all other age groups between and including 12 and 65, give just 2% discount on ticketPrice.

3. A Sample Test-Tube activity

```
public class TracingWhileLoops_3 {
    public static void main(String[] args){

        int i = 0;
        int result = 0;
        int gate = 5;
        int n = 1;

        while(i<gate){

            if(i/4 == 0){
                result = result + 1;
            }
         i = i + n;
        }
    }
}
```

1. Determine the value of result, i/4 and (i<gate) for each iteration of the while loop and complete the table shown below

gate = 5	n =2	i	result	i/4	i<gate
5	2	0	0		
5	2				
5	2				
5	2				
5	2				
5	2				

2. Determine the value of result, i/4 and (i<gate) for each iteration of the while loop and complete the table shown below for a gate = 10 and n = 3. Add more rows if needed.

gate = 10	n =3	i	result	i/4	i<gate
5	2	2	0		
5	2				
5	2				
5	2				
5	2				
5	2				

A Sample Messed-up Code Activity

Problem: Use decision structures to check if a variable **userLetter** is a vowel in the English alphabet. Assume the value of userLetter is already obtained from the user and set to an appropriate data type in each of the following responses. Correct the errors each of the following responses that assumes a given data type for userLetter,

Response 1: userLetter is a String.

```
if (userLetter.equalsIgnoreCase "a"){
System.out.println("Letter is a vowel");
}

if (userLetter.equalsIgnoreCase "e"){
System.out.println("Letter is a vowel");
}

if (userLetter.equalsIgnoreCase "i"){
System.out.println("Letter is a vowel");
}

if (userLetter.equalsIgnoreCase "o"){
System.out.println("Letter is a vowel");
}

if (userLetter.equalsIgnoreCase "u"){
System.out.println("Letter is a vowel");
}

else{
System.out.println("Letter is not a vowel");
}
```

Response 2: userLetter is a char

```
if(user == a){
        System.out.println("It's a Vowel ");
    }
```

```
else if (user == e){
            System.out.println("It's a Vowel ");
        }
else if (user == i){
            System.out.println("It's a Vowel ");
        }
else if (user == o){
            System.out.println("It's a Vowel ");
        }
else if (user == u){
            System.out.println("It's a Vowel ");
        }
else {
           System.out.println("Not a vowel ");
        }
```

Response 3: userLetter is a String and you need to use a || in your if condition

```
if else(letter.equalsIgnoreCase("A||E||I||O||U")){
        System.out.println("you got a vowel");
}
```

Response 4: userLetter is a char and you need to use a || in your if condition

```
if (userLetter = a || e || I || o || u) {
System.out.println("This letter is a vowel.");
else if () {
System.out.println("This letter is not a vowel.");}
```