Towards Improving Student Expectations In Introductory Programming Course With Incrementally Scaffolded Approach

Deepak Dawar
daward@miamioh.edu
Miami University
Hamilton, Ohio

Abstract

Keeping students motivated during an introductory computer programming can be a challenging task. Looking at its varied complexities, many students who are introduced to computer programming for the first time can easily become demotivated. This work looks at the value-expectancy motivational model of student learning, and presents our experiences with a novel instructional delivery interventional technique, introduced and tested over a period of three semesters. Our research question was simple: “Can we affect student motivation, and learning outcomes by using an approach that makes targeted continuous engagement with course material mandatory?” The technique/process was conceived keeping in mind our previous work on similar lines; our in-class teaching experiences; motivational theory; and recent developments in cognitive load theory. The students, instead of writing an assignment and a lab for each module/chapter, were asked to complete one assignment a day, not exceeding four assignments a week. The assignments were incrementally difficult and had to be done almost every day. Students found the approach extremely effective, in spite of having to spend considerable amount of time on assignments. Average final exam scores showed a healthy improvement after the use of this technique.

Keywords: Student motivation, introductory programming, pedagogy, value-expectation, student procrastination, learned helplessness.

1. INTRODUCTION

The landscape of the potential problems faced by novice programmers is vast, and is quite formidable. Teachers with substantial experience in teaching programming, including ourselves, would potentially agree with the above statement. In introductory programming courses, failure rates are high (Allan & Kolesar, 1997; Bennedsen & Caspersen, 2007; Beaubouef & Mason, 2005; Howles, 2009; Kinnunen & Malmi 2006; Mendes et al., 2012; Newman, Gatward, & Poppleton, 1970; Sheard & Hagan, 1998; Watson & Li, 2014), and students can easily become demotivated. One important reason for this demotivation is found in the complex nature of computer programming. The novice programmer has to grapple with multiple domains of learning as suggested in the literature (Davies, 1993; Kim & Lerch, 1997; Rogalski & Samurçay, 1990; Robins, Rountree & Rountree, 2003;). Hence, keeping students motivated is an important part of teaching introductory programming.

Instead of dealing with the multi-faceted motivational aspects of programming directly, we looked at how a student values learning; and what are his/her expectations from that learning. This is derived from the value-expectation theory of motivational design of instruction (Keller, 1983). This theory connects value, expectation, and subsequent motivations as:
Given these theories and challenges, we had to decide which part (expectancy or value) of the motivation model should we try to affect (if there is such a possibility), to improve overall motivation of students, and hence learning outcomes. The value variable in the motivation model is very subjective. There can be myriad reasons why a student may or may not value learning. Fallows & Ahmet (1999), list a set of points regarding value students attach to learning, prominent of which are: 1) philosophical attitude towards learning 2) career aspirations 3) degree of interest in the course etc. A student might find value, and hence may be motivated by multiple factors listed above. We opine that these are very personal beliefs, and it may not be easy to manipulate them in a limited setting of classroom. Therefore, we turn to the expectancy variable in the equation.

Students must believe that they can succeed in the course if they are to be motivated. What are the major causes of student demotivation? There can be many, but the one suspect that we can categorically point towards in our classrooms is high cognitive load. Cognitive load theory (Paas, Renkl, & Brünken, 2010; Sweller, 1988, 1994) deals with the aspects of load placed on working memory while a task is being executed. 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). Hence, high cognitive loads can diminish expectations of a novice programmer leading to a dip in overall motivation, and the value-expectancy model tells us that students must believe that succeed in doing the current assignment, and overall final assessment.

Keeping all these factors and the expectancy model in mind, we designed an intervention that made continuous targeted interaction between the material and students - somewhat mandatory. This approach was designed to influence the expectancy factor in the equation, as this variable seems to be more sensitive to teacher’s or the environment’s influence. 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. This, we opined, would:

- establish a study pattern for students

To design an effective instructional delivery mechanism, we must shed light on what teaching means to the instructor, and what learning means to a student. A student’s level of engagement will depend on their view of activity, and motivation levels. Biggs (1999) provides a general framework regarding conceptions of learning and teaching as a function of three levels. These levels are:

- **Level 1**: Learning as a function of what student is
- **Level 2**: Learning as a function of what teaching is
- **Level 3**: Learning as a function of what activities the student engages in, as a result of the teaching environment

Biggs presents these levels in order of increased complexity with Level 3 being most conducive to learning.

It is imperative to briefly discuss what constitutes a productive teaching climate. McGregor (1960), proposes two competing ideas that can be applied to a workplace and calls them Theory X and Theory Y. Biggs takes these concepts and applies them to academic environment. Theory X assumes that students are unmotivated, and are unwilling to learn. So they must be forced to work hard. Clearly, teacher controls the whole environment, and there is a distrust between the teacher and the student. At the opposite end, Theory Y assumes that students are well motivated, and therefore, must be trusted to work and learn. Assessments should be few, and deadlines must be not enforced strictly. The control somewhat is with the students, and they will respond to this by working voluntarily. In our experience, none of these theories work very well in a classroom. The answer may lie somewhere in the middle.

\[
motivation = expectancy \times value \quad (1)
\]

It follows that if a student feels that the task is worth doing, but finds it impossible to finish, motivation levels are bound to dip (Crego et. al, 2016). Similarly, if a student see no value in learning, if he/she will not be motivated. A teacher or the environment may have a limited affect on some factors, and may have a high impact on others. For instance, it may be quite difficult for the teacher to influence the value variable in the equation i.e. a teacher might have a limited impact on how a student values learning.

...
• improve student’s expectation since the assignments would carry germane cognitive loads
• make them practice programming every almost everyday. This was done keeping in mind the generally accepted notion that constant practice improves the learning outcomes, and as evidenced by psychological studies (Brown & Bennett, 2002; Glover, Ronning & Bruning, 1990; Moors & De Houwer, 2006) done on variable student populations. Constant practice can also make students want to learn more (Kalchman, Moss & Case, 2001) thereby potentially improving the motivation as a whole.

In a series of studies conducted by Rist (1986, 1989, 1995, 2004), and reviewed by Sorva (2012) confirm that one of the main differentiators of students into novice and expert programmers is their constant engagement and experience with learned schemata.

2. METHODOLOGY

This paper builds on the previous work published by Dawar (2020). In that work, students were strictly asked to turn in an assignment a day, and deadlines were more strict. They called it AAAD or 'An Assignment A Day Scaffolded Approach'. This paper builds on that work in the following terms.

1. It refines the AAAD approach by dynamically adjusting deadlines while still mandating most assignments to be submitted within a day.
2. Looks into the relationship of altered cognitive load and student expectations.
3. Provides additional data to support the conclusions drawn in the previous work.
4. Provides a framework for future work in this direction.

Our method rests on three pillars as shown below in Fig 1.

![Figure 1: Teaching Intervention](https://www.iscap.info)

The study was conducted over three semesters. The control group (C1) data was collected in the first semester (Fall 2018). 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 work per week, with quizzes at the end of the module/chapter.

In the next semester (Spring 2019), the experimental group (E1) was administered the interventional approach, and pertinent 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 (see Appendix C) was conducted with this experimental group to measure how well this approach was received by the students. The experiment was again repeated in the third semester (Fall 2019) with another experimental group E2. We followed the exact same procedures for E2 that were followed for E1 with slight deadline modifications especially for full time working students. All other factors like quizzes, projects etc. remain the same for control and experimental groups.

The number of students in C1, E1, and E2 were 20, 22, and 21 respectively. One student from C1 and three students from E2 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 C1 had 12 IT/CSE majors and 8 non IT/CSE students. The experimental group E1 had 13 IT/CSE, and 9 non IT/CSE majors. E2 had 12 IT/CSE, and 8 non IT/CSE majors respectively. So the class composition of all groups compared was fairly similar with C1, E1, and E2 having about 40%, 41%, and 40% non IT/CSE majors respectively. This relatively similar class composition gives us some level of confidence about the experimental set up.

Administering the right cognitive load is crucial to success of this intervention. As can be inferred 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. The task load belongs to the chapter that concerns itself with “method writing” in JAVA. This was to be delivered as an approximately eight-day module with classroom practice labs (non-graded), five assignments, and a quiz at the end. Detailed descriptions of these assignments are included in Appendix B.

Comparison
Since the experimental groups (E1 and E2) had to do many more assignments (at least 4 more assignments per module), an equitable comparison between the control and experimental groups was a challenge.

We decided that the comparison of the last summative assignment given to the experimental group(s) with the usual single assignment per module 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 would have had more exposure to the concepts since they would have submitted a series of assignments before attempting the final assignment.

We assessed the following metrics for both groups, and for each assignment compared.

- assignments submitted late
- assignments not submitted

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:

- module wise quiz scores
- final exam scores

3. RESULTS
We divided our analyses into two parts - inter and intra group. Inter group analyses compared the control (C1) with experimental groups (E1, E2), and intra group compared/analyzed the results of the experimental groups (E1, E2) only.

Inter Group Analyses
The control group did only one assignment per week whereas the experimental groups did several leading up to the last assignment of the module. We compared the statistics of the last module assignment of the experimental group with the usual weekly assignment of the control group.
The data collected lays out some interesting points. The experimental groups, at an anecdotal level, showed a greater inclination to submit the final assignment as compared to the control group. Bear in mind that the experimental group students - by the time they submit the final assignment - have already submitted multiple assignments on module topics leading up to the last assignments. The 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 groups, the late submission rate is higher. Late submissions in both control and experimental groups 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 groups even if that meant submitting it late. This is evident from the fact that there are more late submissions in experimental groups than no submissions. The trend is reverse in the control group. This is to reiterate that the data presented here for experimental groups 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 be no significant difference in the quiz performance for the groups, though the standard deviation in the experimental groups 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 presents the data.

<table>
<thead>
<tr>
<th>Group</th>
<th>Average Final Quiz Score</th>
<th>Average JAVA Program Score</th>
<th>Cumulative Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>66%</td>
<td>51%</td>
<td>56%</td>
</tr>
<tr>
<td>E1</td>
<td>74%</td>
<td>71%</td>
<td>72%</td>
</tr>
<tr>
<td>E2</td>
<td>78%</td>
<td>74%</td>
<td>75%</td>
</tr>
</tbody>
</table>

Table 5: Final exam score for all groups

It is quite interesting to note that while there was no significant difference between module quiz scores, the experimental groups performed much better in the final exam. Even though the gains in the final quiz are marginal, the experimental groups outperformed the control group by 20 percentage points or more in JAVA program writing. The overall cumulative improvement in final exam mean score was 16%, and 19% for E1 and E2 respectively. These numbers may insinuate that – for the experimental groups – 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.

**Intra Group Analyses**

Table 6 and 7 present detailed non-submission data for E1 and E2 respectively. 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.

Table 8 and 9 represent the late submission data for E1 and E2, respectively. Table 10 and 11 present a cumulative summary of the assignments for E1 and E2 respectively. Cumulatively, for both experimental groups, 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.
One of the questions asked the students about how they felt about the utility and effectiveness of this intervention in completing the course satisfactorily. A surprising 90% of the students in E1 and 84% in E2 answered that they felt positive/better about using this technique while 10% in E1, and 9% in E2 reported that they felt slightly worse while working with this technique. Another question asked the students about the utility of doing a daily assignment in learning computer programming. A whopping 100% of the students in both E1 and E2 felt that it is useful. This gives us some confidence to assert that given the right cognitive load and environment, students do see potential value in constant practice for learning programming.

Another important question asked the students about their choice between the novel instructional technique and the normal course delivery mechanism of doing one assignment per week. 96% in E1, and 76% of students in E2 preferred the novel technique. On an aggregate level, 88% of the students said that they would prefer working everyday, 6% preferred orthodox course delivery, and 6% showed no preference. Hence, the students overwhelmingly choose working everyday as a mode of course delivery over our normal delivery method. This, we believe, is a very important piece of feedback for us. Students were also asked about their stress levels regarding doing so many assignments. A cumulative 45% of the students answered that working everyday on assignments made it easy for them to manage stress.

Students remarked that the process made it easy to manage overall stress as the assignments were gradually increasing in difficulty. 39% said it increased their stress levels as they had to do many more assignments, and 15% choose that it made no difference.

The efficacy of this intervention cannot be generalized with such a small sample space but the initial results do reveal some interesting insights. Many students seem to find working on incrementally difficult assignments beneficial, even if it means working more time than usual. According to the assignment data collected and student responses on the survey, most students show an inclination towards practicing more, as long as the cognitive load is manageable. This is evident 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.

Late submissions were allowed with reduced credit, and cumulative late submission rate stands at about 10.5%, and 11%.

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 later stage assignments.

### 4. COURSE SURVEY AND DISCUSSION

An end of course survey was conducted for both E1 and E2. Number of participants were 22, and 13 respectively i.e. 35 students in total. The questions were primarily centered around the potential impact of high number of assignments on their motivation, stress levels, and their choice between the instructional intervention and the orthodox method of single assignment per module used at our department. The full survey is listed in appendix C.
Confirming our expectations, the instances of no-submission and late-submission rise during these modules. Overall, this technique, appears to successfully increase student engagement with the course.

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 demotivation, and may even exacerbate the de-motivational factor this technique was designed to mitigate. Results and responses, however, show that the technique successfully navigated these roadblocks.

A 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.

Another important aspect was the continual and immediate presence of instructor and tutor support. Without this perennial support, this technique may be rendered ineffective very quickly. Our experience in a more traditional approach is that about 50%-60% of the students asked questions on the day the assignments were due. Since students have a due date almost every day of the week, it 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. The daily deadline were especially difficult for the full time working students. For them, as evidenced by comments in the survey, it was difficult to schedule time every day to finish the assignments.

5. CONCLUSION AND FUTURE WORK

Students in both experimental sections of our introductory programming course agreed that working on incrementally difficult assignments everyday added value to their process of learning computer programming. It helped them practice consistently, thereby improving their enthusiasm about the course and programming. Though there was no significant differences in the individual chapter quiz scores between the control and experimental groups, the experimental groups performed much better in the final exam. At an anecdotal level, it seems that it may be possible to affect the motivation levels of students using this intervention. The end of course survey responses indicate that though the technique was very well received.

It would be too premature to consider the intervention as a success given the significant challenges this techniques entails. Firstly, grading a large number of assignments, and providing high volume of feedback is resource intensive. Hence, an automatic grader may be required to speed things up. Continuous tutor support is also required to help stem student frustration, and to give them the feeling that help is always available.

![Figure 2: Incrementally Scaffolded System: An Abstraction](image)

To mitigate the load on the instructor, tutor/grader and students while maintaining the integrity of the technique, we envisage coupling an automatic grading system with an artificial tutor bot, capable of answering basic questions about the course, assignments, and simple concepts of programming. An abstract schemata of this system is shown in Figure 2. We are encouraged by the initial results of this study, and the promise of future research.

6. REFERENCES


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.


## APPENDIX A

### Table 1: Increment in cognitive load with time

<table>
<thead>
<tr>
<th>Assignment No.</th>
<th>Description</th>
<th>Concepts Tested</th>
<th>Cognitive Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write a method printS that takes a string as an input and prints it to the console.</td>
<td>Rudimentary method writing.</td>
<td>Low</td>
</tr>
<tr>
<td>2</td>
<td>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.</td>
<td>Method writing, method calling, method modification.</td>
<td>Low</td>
</tr>
<tr>
<td>3</td>
<td>Reuse printS to print a user entered string ( n \times n ) times i.e a square with each element as the string</td>
<td>User input, loops, method writing, method calling</td>
<td>Medium</td>
</tr>
<tr>
<td>4</td>
<td>Reuse printS method to print a right angle triangle in terms of user entered string</td>
<td>User input, loops, method writing, method calling, Problem solving</td>
<td>Medium</td>
</tr>
<tr>
<td>5</td>
<td>Reuse printS to print a pyramid in terms of user entered string</td>
<td>User input, loops, method writing, method calling, Problem solving</td>
<td>High</td>
</tr>
</tbody>
</table>
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 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.
APPENDIX C

CSE 174: Student experiences with multiple assignments

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?

<table>
<thead>
<tr>
<th>Definitely yes</th>
<th>Probably yes</th>
<th>May be</th>
<th>Probably not</th>
<th>Definitely not</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Definitely yes</th>
<th>Probably yes</th>
<th>May be</th>
<th>Probably not</th>
<th>Definitely not</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Extremely easy</th>
<th>Moderately easy</th>
<th>Slightly easy nor difficult</th>
<th>Slightly difficult</th>
<th>Moderately difficult</th>
<th>Extremely difficult</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Extremely easy</th>
<th>Moderately easy</th>
<th>Slightly easy nor difficult</th>
<th>Slightly difficult</th>
<th>Moderately difficult</th>
<th>Extremely difficult</th>
</tr>
</thead>
</table>
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
- It made me practice moderately better
- Slightly better
- I would have practiced a lot regardless of this format
- It made me practice much better
- It made me practice less
<table>
<thead>
<tr>
<th>Outcome 1: Use and describe a contemporary programming language and programming environment (IDE) like Dr. Java.</th>
<th>Extremely well</th>
<th>Very well</th>
<th>Moderately well</th>
<th>Slightly well</th>
<th>Not well at all</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outcome 2: Identify and eliminate errors in programs</th>
<th>Extremely well</th>
<th>Very well</th>
<th>Moderately well</th>
<th>Slightly well</th>
<th>Not well at all</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>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</th>
<th>Extremely well</th>
<th>Very well</th>
<th>Moderately well</th>
<th>Slightly well</th>
<th>Not well at all</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outcome 4: Solve programming problems using a procedural approach i.e. divide your program into methods</th>
<th>Extremely well</th>
<th>Very well</th>
<th>Moderately well</th>
<th>Slightly well</th>
<th>Not well at all</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outcome 5: Describe, trace, and implement basic algorithms like linear search, binary search etc.</th>
<th>Extremely well</th>
<th>Very well</th>
<th>Moderately well</th>
<th>Slightly well</th>
<th>Not well at all</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outcome 6: Apply and communicate information that they read from technical sources such as APIs like Scanner etc.</th>
<th>Extremely well</th>
<th>Very well</th>
<th>Moderately well</th>
<th>Slightly well</th>
<th>Not well at all</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>