Coding the Coders: A Qualitative Investigation of Students' Commenting Patterns

Dr. Mahnas Jean Mohammadi-Aragh, Mississippi State University

Dr. Jean Mohammadi-Aragh is an assistant professor in the Department of Electrical and Computer Engineering at Mississippi State University. Dr. Mohammadi-Aragh investigates the formation of engineers during their undergraduate degree program, and the use of computing to measure and support that formation. She earned her Ph.D. in Engineering Education from Virginia Tech. In 2013, Dr. Mohammadi-Aragh was honored as a promising new engineering education researcher when she was selected as an ASEE Educational Research and Methods Division Apprentice Faculty.

Ms. Phyllis J. Beck, Mississippi State University
Ms. Amy K. Barton, Mississippi State University

Amy Barton is Technical Writing Instructor in the Shackouls Technical Communication Program at Mississippi State University. In 2013, she was inducted into the Academy of Distinguished Teachers for the Bagley College of Engineering. She is an active member of the Southeastern Section of ASEE. Her research focuses on incorporating writing to learn strategies into courses across the curriculum.

Dr. Donna Reese, Mississippi State University

Donna Reese is currently a professor of Computer Science and Engineering at Mississippi State University. She served as head of the department from 2010 to 2016. Prior to that she served for six years as associate dean in the Bagley College of Engineering. Her research interests are in recruitment and retention of underrepresented groups in computing and engineering fields.

Dr. Bryan A. Jones, Mississippi State University

Bryan A. Jones received the B.S.E.E. and M.S. degrees in electrical engineering from Rice University, Houston, TX, in 1995 and 2002, respectively, and the Ph.D. degree in electrical engineering from Clemson University, Clemson, SC, in 2005. He is currently an Associate Professor at Mississippi State University, Mississippi State, MS.

From 1996 to 2000, he was a Hardware Design Engineer with Compaq, where he specialized in board layout for high-availability redundant array of independent disks (RAID) controllers. His research interests include engineering education, robotics, and literate programming.

Ms. Monika Jankun-Kelly, Mississippi State University

Monika Jankun-Kelly has taught introductory and intermediate computer science courses at Mississippi State University for several years.

Coding the Coders:
Creating a Qualitative Codebook for Source Code Comments

1. Introduction

The struggles of novice programmers are significant and well documented, with 30-50% of students failing their first programming course [1]. Studies have examined many possible factors, and often focusing on identifying the areas of aptitude or student characteristics that are linked to innate programming ability [2]. The limitation of examining factors in the context of innate ability, however, is that they do not adequately address the wide range of abilities and challenges represented in a typical first-year programming course. Additionally, the limitation to designing pedagogy with the idea that a student is either born a programmer or not leaves students with little control over their success or failure within a course. In contrast, pedagogical approaches that encourage students to monitor their own learning can help the student recognize their ability and to make adjustments as needed. Specifically, students who effectively employ metacognitive strategies, such as reflection and self-assessment, are more likely to master the problem solving skills that are essential to programming success [3].

Writing to learn (WTL) activities promote metacognition in any discipline. Based on the idea that writing is a visual representation of thinking [4], WTL activities are usually short, low-stakes writing assignments that are designed to promote reflection, analysis, synthesis, and deeper understanding of course material. When integrated into a problem-solving assignment, such as a programming lab, WTL prompts allow students to think about the choices they are making and the reasons for those choices. When employed at the end of an assignment, reflection questions encourage students to recognize what they learned, identify errors, and consider different choices they might make in the future. Throughout an entire course, students' writings become an artifact of the changes and growth that accompany learning and provide instructors with a rare insight into students' learning processes.

Our team is currently investigating how intermingled writing and coding can improve the process of learning to program. We have incorporated WTL strategies into introductory computer programming laboratory assignments and are comparing student work from those laboratories with student work from traditional laboratories. In order to minimize additional work for the WTL students, our initial investigation has focused on examining existing writing in the form of source code comments. The research discussed in this paper focuses on answering the following two research questions: *RQ1) What do source code comments tell us about novice programmers' thinking processes while coding? RQ2) How are students visually organizing their source code?*

2. Background

Kolb's experiential learning cycle of experience, reflection, conceptualization, and experimentation has been used to support deeper learning in computer science instruction [5]. Effective learning requires a learner to go through all four stages of the cycle. Because reflection is essential to the cycle, reflective writing has been used as a tool in computer science classrooms. Moore [6] required weekly journals and noted that the responses revealed direct

evidence of student learning processes, which were supported by improved achievement. Additionally, Moore found that students were able to use their work to solve similar issues in future programs. Beyond the classroom, George [7] advocates the reflective journal as a way to foster reflection in software engineering practice. Reflective journals can be responses to specific prompts or unstructured diary entries [8]. Van Wyck [9] and Ladd [10] argue that programming *is* writing and therefore should be taught using the same metacognitive strategies of reflection and revision. By requiring programming portfolios, they emphasize the iterative learning process while prompting students to consider the clarity of their thinking at each stage of the process. Ladd asserts that this clarity of thinking is also essential for best practices in programming, noting that a common flaw in introductory programming courses is the failure to stress that "writing computer programs is writing to communicate with a human and a machine audience" [10, p. 57]. By communicating their thinking processes in writing, students are more aware of the human side of this interaction. The writing also reflects their progress through the learning cycle of experience, reflection, conceptualization, and experimentation.

If success in learning to program depends on building knowledge and independence, self-regulation is critical to students' progress. Novice programmers can struggle when attempting to assess their own mastery. A 2005 international survey of 500 programming students and teachers revealed that students tend to overestimate their understanding [11]. Papadopoulos et al. studied students' ability to process complex information by requiring one group of students to simply think about their responses to questions and the other group to write reflective responses [12]. They found a significant difference in student assessments of learning, with knowledge acquisition favoring the writing group. In the thinking group, students had false self-awareness of their learning process, and would have benefited from providing their answers in writing. Thus, a primary benefit of WTL strategies is the ability to externalize, or make visible, the thinking process. For students, the implication is that the act of writing makes their learning visible to themselves, which clarifies and deepens thinking.

While others have shown that writing reflections after programming supports learning, we are investigating the impact of fully integrating explanations and reflections into the act of programming. In our WTL implementation, we follow the principles of Knuth's literate programming paradigm [13], which views programming as authoring a document that happens to contain source code.

3. Methods

To investigate intermingled writing and coding, we modified the instructions to traditional laboratory (TRAD) assignments for an introductory computer science course. The traditional instructions required students to 1) create a design using pseudo-code, 2) code the design, and 3) write a final report. The WTL assignments were identical, except that students were instructed to create their code and report at the same time using a literate programming tool, which allowed students to view the source code and the typeset document together in a split screen view.

To examine students thinking processes, we employed qualitative methods, which are appropriate for understanding phenomena from the often complex viewpoint of someone else. To develop our qualitative codebook, we used systematic open coding techniques outlined by

Strauss and Corbin [14] in two-phases to discover categories. In the first phase, we addressed RQ1 and focused on identifying categories relevant to students' thinking processes (see Section 4). The second phase of qualitative coding focused on RQ2 and identifying categories related to visual organization (see Section 5). In both phases of analysis, we used exploratory coding, as opposed to using predetermined codes. For example, while we anticipated reflective comments within source code based on the WTL instructions, we did not assume there were reflective source code comments, and instead let the data determine the qualitative codes. We examined multiple weeks of laboratory assignments submitted by 12 TRAD and 13 WTL paired-programming groups (50 students total). Course topics that these assignments covered included simple branching, if, if/else, Boolean algebra, logic operators (and, or, not), while loops, and flow charts. After creating our initial qualitative codebook, we tested it by coding a laboratory assignment that was not part of the original assignment set (see Section 6). We included source code from two additional laboratories resulting in 23 TRAD and 20 WTL paired-programming groups (86 students total). The assignment used in this analysis focused on for loops and nested loops. During all phases of analysis, we used qualitative data analysis software, MaxQDA, for qualitative coding of source code comments.

4. Coding Students' Thinking Processes

To analyze students' thinking processes, we examined comments within source code only; we omitted all other portions of the laboratory assignment from analysis, including the reflection portions at the end of the laboratory reports. We analyzed each source code comment line-by-line to determine if it contained meaningful information and the type of thinking it represents (e.g., explaining an action, providing justification for a design choice). As we examined the source code comments, we identified five types of comments: *literal*, *conceptual*, *reflective*, *organizational*, and *insufficient*. In addition, a sixth code, *none*, was used when there are no comments within the program.

4.1 Literal Comments

A *literal* comment simply restates the source code or calculation in English. The comment adds no additional meaning to the understanding of the source code. Literal comments give no reasons why certain choices were made. Examples of literal comments are shown in Fig. 1. We observed a large percentage of literal comments within the source code. We hypothesize this is related to the use of pseudo-code as a design tool, which is strongly encouraged in the introductory programming courses we studied.

```
72  # If days left is less than 3 and greater than or equal to 2
73  if remainingDays < 3 :
74      if remainingDays >= 2 :
75  # Display the water status as low
76          waterStatus = 'Low'
```

Figure 1: *Literal* comment example (shaded green)

4.2 Conceptual Comments

A *conceptual* comment conveys an understanding of how the source code works by explaining source code functionality. Conceptual comments do not simply restate the source code in English but add additional understanding that will improve readability and comprehension of the

program from an outside perspective. An example conceptual comment is shown in Fig. 2. It is important to emphasize that if the comment in Fig. 2 had read, "Calculate the days remaining," it would have been classified as literal. By adding "before the water runs out," the student shifts this comment into the conceptual domain by adding additional information about the purpose of the calculation. We have noticed that for a few comments it can be challenging to differentiate between a conceptual or literal classification. In these cases, we focus on 1) whether or not it is possible to remove a few words and still have a useful comment, and 2) whether or not the removed words added understanding to the original comment. If the response is affirmative for both those items, the comment is classified as conceptual.

```
62  #
63  # Calculate the days remaining before the water runs out
64  remainingDays = int( tankVolume / ( waterPerPerson * numResidents ) )
```

Figure 2: *Conceptual* comment example (shaded blue)

4.3 Reflective Comments

A *reflective* comment is representative of internal dialogue and explains why the student took a particular approach to solving a problem. For example, a student may explain why they chose a certain programming library (Fig. 3). Reflective comments more clearly reveal students programming metacognition.

```
89  #       leditl Before the actual code we need to import math so that we can use the
    cos function and the sqrt function later in the code.
90  import math
```

Figure 3: *Reflective* comment example (shaded magenta)

4.4 Organizational Comments

An *organizational* comment is used to communicate the organizational structure of code. Organizational comments were typically used to indicate the beginning of a new section of code with separate functionally. This demonstrates that a pair is actively attempting to organize code in the way that they are perceiving it. Originally, we included organizational comments with reflective comments but we chose to separate them in order to differentiate between internal dialogue and organizational dialogue. Fig. 4 illustrates an example of an organization comment, as well as two literal (green) and five conceptual comments (blue).

```
49  # **Calculations**
50  #
51  # These calculations are performed inside of a while loop.
52  while i <= D:
53  # The volume in cubic meters.
54    Vm = L*(R**2*acos((R-h)/R)-(R-h)*sqrt(2*R*h-h**2))
55  # The volume converted into litres
56    VL = Vm*1000
57  # The total time left based on the number of people.
58    Time = VL/(People*Use)
59  # The time component in days.
60    TimeD = floor(Time)
61  # The remainder of the time in hours.
62    TimeH = (Time - TimeD)*24
63  # Then print these findings in a formatted table.
64    print(format(i,'20.2f'),'|', format(VL,'20.2f'),'|', format(TimeD,'10.0f'),'|', format(TimeH,'10.2f'))
65    i = i + 0.1
66    h = h - 0.1
```

Figure 4: *Organizational* comment example (shaded red)

4.5 Insufficient Comments

A program comment is *insufficient* if it is either too short to warrant a more complex classification, or the comment adds no value even if it is verbose. We do not currently know if the terseness is due to a lack of understanding, laziness in commenting, or mastery of content resulting in a belief that it is unnecessary to comment. However, the reasoning is not critical at this juncture; the comments exist, so our qualitative codebook must include a code for classifying them. Fig. 5 illustrates an insufficient comment that is both confusing and provides no additional value.

```
60    # applies change in i to h
61    h=d-i
```

Figure 5: *Insufficient* comment example (shaded yellow)

5. Coding Students' Visual Organization

Students' visual organization style is a representation of the use of whitespace, comments, and blocks of code to organize their overall source code. Five qualitative codes emerged from our analysis of visual organization strategy: *unitization*, *every-line*, *block-level*, *insufficient*, and *none*. Unlike coding students' thinking process, which was coded per individual comment, visual organization was coded based on an analysis of the entire source code. When examining thinking processes, we often saw multiple types of comments (e.g., conceptual and literal) within a single source code. However, with visual organization, it was more typical for students to employ a single strategy throughout the entire source code.

5.1 Unitization

Comments were coded as *unitization* when they organized source code into logical sections. We define a logical section as a group of lines of code that are related to a particular function or

action within the source code. Fig. 6 provides an example of unitization with literal (green) and conceptual (blue) comments.

```
28  # - Create the header for the table
29  print(format("Measured depth ","20s"),("|"),format("Tank volume","20s"),("|"),format("Water","20s"),("|"),("Remai
30  print(format("in m","20s"),("|"),format("in L","20s"),("|"),format("Days","20s"),("|"),("Hours"))
31  print(format("--------------------","20s"),("+"),format("--------------------","20s"),("+"),format("--------------------","20s"),("

32  # - Create loop that runs through all the deiffernet depths by .1 meter.
33  while depth <= 2.00:
34      height=diameter-depth
35
36  # - Calculate volume; math: V=L(R^2 \cos^{-1}\frac{R-2}{R}-{R-h}\sqrt{2Rh-h^2}}
37      tank_meters=(length*((radius**2)*(acos((radius-height)/radius))-((radius-height)*sqrt(2*radius*height-height*
38      tank_volume=tank_meters*1000
39      water_per_day= (tank_volume/(residents*2))
40      day_total=(floor(water_per_day))
41      hour_total=(water_per_day-day_total)*24
42
43  # - format and print the rest of the table
44      print(format(depth,">20.2f"),("|"),format(tank_volume,"20.2f"),("|"),format(day_total,"20d"),("|"),format(hour_to
45      depth+=.1
```

Figure 6: A code snippet illustrating *unitization* organization with three *literal* comments (green) and one *conceptual* comment (blue)

5.2 Every-line

An *every-line* visual organization style is characterized by comments that precede or sit side-by-side nearly every line of code in a program and is often an over utilization of code commenting. There may or may not be an effective use of white-space. Every-line commenting is thought to be a natural offspring of students being encouraged to use pseudo-code for comments. Fig. 7 provides an example of every-line with literal (green) and conceptual (blue) comments.

```
# Explain the code to the user
print( 'This program will help you find thetime remaining until the water in a water tank is use
print( '' )
# Import math for the functions you will use later
import math
# Ask for the length of the tank, in meters
tankLength = float( input( 'Length of water tank, in meters: ' ) )
# \
#
# Ask for the diameter of the water tank, in meters
tankDiameter = float( input( 'Diameter of the water tank, in meters: ' ) )
tankHeight = tankDiameter
tankHeightInitial = tankHeight
# \
#
# Ask for the number of residents the tank will serve
numResidents = int( input( 'Number of residents this tank will serve: ' ) )
# \
#
# Each person uses a constant 2 liters of water per day
waterPerPerson = int( 2 )
# \
# Divide the diameter by 2 to find the radius
tankRadius = tankDiameter / 2
# \
#
# Calculate the volume when no water has been used, or height is at max
tankVolume = ( ( tankLength * ( ( tankRadius ** 2 ) * ( math.acos( ( ( tankRadius - tankHeight )
- ( tankRadius - tankHeight ) * ( math.sqrt( ( 2 * tankRadius * tankHeight ) - ( tankHeight ** 2
#
# Calculate the days remaining before the water runs out
remainingDays = int( tankVolume / ( waterPerPerson * numResidents ) )
# \
#
# Multiply the remainder by 24 to find the hours remaining
remainingHours = float( ( ( tankVolume / ( waterPerPerson * numResidents ) ) - remainingDays ) *
```

Figure 7: A code snippet illustrating *every-line* organization with seven *literal* comments (green) and three *conceptual* comments (blue)

5.3 Block-level

Block-level commenting occurs when blocks of comments are preceded by large blocks of code that are too large to be considered unitization and the code could benefit from being broken down into smaller units of organization. This is apparent in the way one would break up code to be used in a function. If the unit can be divided into more than one function, a block-level classification may be appropriate. Fig. 8 provides an example of block-level with conceptual (blue) and reflective (magenta) comments.

```
# - If the patient has a systolic blood pressure is less than 90 mmHg,
#   the user should be told the patient needs immediate medical attention.
#   If not, the user will be asked about the patient's age. This information
#   will be used by the program to determine if the patient is an infant
#   (less than a year) or not an infant. This will set two pathways to
#   determine the respiratory rate decision.
else:
    systolicBloodpressure = input("What is your systolic blood pressure?")
    systolicBloodpressure = int(systolicBloodpressure)
    if systolicBloodpressure <= 90:
        print("Recieve Immediate High Priority, Medical Attention")
# - In the infant path way, if the respiratory rate is less 20 breaths per min
#   then the user will be told to give immediate care. In the other pathway,
#   if the respiratory rate is less than 10 breaths per min or greater than
#   29 breaths per min then the user will be told to give immediate care.
    else:
        Age = input("What is your age in years?")
        Age = float(Age)
        if Age < 1:
            infantBreathe = input("What is the infant's respiratory rate?")
            infantBreathe = int(infantBreathe)
            if infantBreathe <= 20:
                print("Recieve Immediate High Priority, Medical Attention")
        if Age >= 1:
            Adultbreathe = input("What is your breathing rate?")
            Adultbreathe = int(Adultbreathe)
            if Adultbreathe <= 10:
                print("Recieve Immediate High Priority, Medical Attention")
            else:
                if Adultbreathe >= 29:
                    print("Recieve Immediate High Priority, Medical Attention")
```

Figure 8: A code snippet illustrating *block-level* organization with two *conceptual* comments (blue) and one *reflective* comment (magenta)

4.5 Insufficient and None

Insufficient organization occurs when there are too few comments within the source code to determine a meaningful organizational classification, or the source code is disorganized. The source code has too much, too little, or inconsistently used white space and there is little to no commenting. With these qualities present, an intentional organization strategy is not apparent. Fig. 9 provides an example of insufficient organization with insufficient (yellow) and literal (green) comments.

An organizational code of none occurs when there are no comments or whitespace in the source code. As opposed to insufficient, which does contain comments but has a difficult-to-discern organizational strategy, the none classification means there is no visual organization strategy. In combination with no comments, the typical source code for none contained either no whitespace or an odd use of white space.

```
23  from math import acos, sqrt, floor
24  # Ask the user for tank length
25  l = int( input( "Length of the water tank, in meters: " ))
26  # Ask the user for tank diameter
27  d = float( input( "Diameter of the water tank, in meters: " ))
28  r = d/2
29  # Ask the user for the number of residents
30  numberOfResidents = int( input( "Number of residents this tank will serve: " ))
31  #
32  # create a table using loops
33  print ( format ( 'Measured Depth' , '15' ), 'l ', end = '')
34  print ( format ( 'Tank Volume' , '15' ), 'l ', end = '')
35  print ( format ( 'Water Remaining' , '16' ))

36  print ( format ( 'in m' , '15' ), 'l ', end = '' )
37  print ( format ( 'in L' , '15' ), 'l ', end = '' )
38  print ( format ( 'Days' , '8' ), 'l ', end = '' )
39  print ( format ( 'Hours', '8' ))

40  print ( '---------------+----------------+----------+-------' )
41  height = 0
42  num = int(d * 10)
43  for h in range (0, num) :
44      tankVolume = 1000 * l* (r**2 * acos ((r-d)/r) - (r-d) * sqrt (2*r*d-d**2))
45      #print ('')
46      #print ( tankVolume )
47      days = float( tankVolume / (numberOfResidents*2))
48      #print ( days )
49      totalHours = days * 24
50      #print( totalHours )
51      daysRounded = totalHours//24
52      #print (daysRounded)
53      remainingHours = totalHours - ( daysRounded * 24)
54      #print (remainingHours )
55      d = d - 0.1
56      print ( format ( height, '15.2f' ), 'l', format ( tankVolume, '15.2f' ), 'l',
57          format( daysRounded, '8.2f' ), 'l', format ( remainingHours, '8.2f' ))
58      height = height + 0.1
59
```

Figure 9: A code snippet illustrating insufficient organization with three literal comments (green) and one insufficient comment (yellow)

6. Initial Analysis of Students' Commenting Patterns

We tested the codebook by analyzing source code from a laboratory assignment for 86 students completing an introductory programming course. We selected an assignment that was completed beyond the mid-point of the semester to ensure students were familiar with the instructions used

in their assigned laboratory, TRAD or WTL. Students completed the assignment in pairs, producing 43 assignments for analysis (23 TRAD and 20 WTL). One WTL and two TRAD assignments had no comments and were excluded from the analysis.

We observed two promising outcomes when applying our codebook. First, the inclusion of new TRAD and WTL laboratory sections allowed us to analyze the robustness of our codebook. While coding thinking processes and visual organization, we observed no new commenting patterns that required new qualitative codes. Second, WTL assignments contained more comments than TRAD assignments (Table 1) and those comments were used to break code into logical sections (*unitization*) more often (Table 2). Additionally, students in WTL sections were more likely to utilize an organizational strategy than their peers in TRAD sections (16/20 versus 15/23).

Table 1: Number of comments per assignment

Metric	TRAD	WTL
Average number of comments	14.2	15.8
Median number of comments	12	14

Table 2: Results of Coding Commenting Style

Commenting Style	TRAD	WTL
Unitization	4	7
Every-line	11	9
Block-level	0	0
Insufficient	6	3
None	2	1

We observed different thinking processes between TRAD and WTL sections but have not fully examined these differences. While all 15 TRAD source codes contained at least one conceptual comment, only 15 of 16 WTL source codes contained at least one conceptual comment. However, the WTL source code with no conceptual comments contained 25% reflective comments. We did observe that students in the TRAD instruction laboratories more often favored a literal/every-line or a conceptual/every-line commenting style, while students in the WTL laboratories commonly used a literal/every-line or a conceptual/unitization style.

6.1 Limitation

We caution against applying the findings from the initial application of our codebook too broadly. While we are motivated by our initial results, it is important to look across multiple laboratory assignments to determine if the trends and themes we have observed continue to appear. Instead, the analysis herein should serve as a snapshot with further analysis investigating changes in thinking processes over time, and linking thinking processes and organizational patterns to direct learning assessments. We were unable to do either of those analysis at this point due to a research limitation with the structure of the laboratories: the laboratories are paired-programming laboratories where the groups change from assignment to assignment. Thus, it was not possible to track the progression of a single student's thinking process or organizational strategy based off laboratory source code comments alone. We are in the process

of obtaining other artifacts, such as individual homework assignments, in order to more fully analyze the differences in the approaches of TRAD and WTL students.

Another limitation involves generating a summary classification of students' thinking processes. Initially, we attempted to classify labs as either very good, good, average, or poor based on a point system using our qualitative codes. Each qualitative code was given a weighted value. Then, a laboratory submission was assigned a score based on a series of thresholds related to the number of source code comments per qualitaive code. However, we observed that this method negatively affected our view of certain programming approaches. For example, the point system appeared to penalize a lab for using pseudo-code when pseudo-code was recommended as good practice in the course lectures. We chose to drop the point system and recommend restraint from viewing particular qualitative codes as good or bad at this point in our process. Ultimately, an effective source code solution may incorporate a variety of comment styles, and additional analysis is needed before linking our qualitative codes to a system for evaluating student assignments.

7. Conclusions and Future Work

In summary, we have created and applied a qualitative codebook to classify comments within computer programs. The qualitative codebook allows for classification of both students' thinking processes and their visual organizational pattern. By applying the codebook to classify students' programming assignments, we found patterns that indicate students receiving instructions that incorporate WTL principles may be more metacognitive while creating source code. We are continuing to classify students' programming assignments to validate and strengthen these initial findings.

We note that others have analyzed and categorized source code comments previously. However, they have focused on quality (e.g., [15-17]) and usefulness (e.g., [18-19]) of the comments, particularly with regards to code documentation or bug detection efforts. Our analysis categorizes comments in order to understand the underlying thinking processes and organizational strategies employed by novice programmers. Our analysis focuses on understanding how we can more effectively help novices develop as programmers, as opposed to analyzing comments to improve software engineering processes.

This paper presents our initial effort in examining how intermingled writing and coding can improve programming pedagogy. Currently, we are in the process of investigating the relationship between comment classifications and student performance in the studied introductory course and plan to track student success through subsequent programming courses as well. As we begin to look at student performance, we are utilizing McGill and Volet's conceptual framework for analyzing student programs and diagnosing deficiencies [20]. This framework links three types of programming knowledge (syntactic, conceptual, and strategic) with the view of knowledge from cognitive psychology (declarative, procedural, and conditional). We believe using this framework to link our qualitative codebook with programming knowledge will allow us to further evaluate students' thinking processes and the changes in programming knowledge over time. Furthermore, in addition to classifying more student assignments with our codebook, we plan to conduct interviews and think-aloud

experiments with students in order to understand thinking processes and source code commenting from their individual perspectives. For instance, it is possible that a classification of *literal* indicates that students back-filled comments after coding in order to meet class assignment requirements as opposed to our initial belief that students are using pseudo-code from the design phase. By implementing these future research plans, we will continue our examination of the impact of WTL strategies in introductory computer science courses with the end goal of improving programming pedagogy and more effectively helping students learn to program.

7. References

[1] C.Watson, and F. W. Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, 39-44.

[2] R. Hoda and P. Andreae. 2014. It's not them, it's us! Why computer science fails to impress many first years. In *Proceedings of the 16th Australasian Computing Education Conference*, 158-162.

[3] S. Bergin, R. Reilly and D. Traynor. 2005. Examining the role of self-regulated learning on introductory programming performance. In *Proceedings of the First International Workshop on Computing Education Research*, 81-86.

[4] J. Emig. 1977. Writing as a mode of learning. *College Composition and Communication*, 28, 122-128.

[5] E. Crowley. 2004. Experiential learning and security lab design. In *Proceedings of the SIGITE 2004 Conference*, (SIGITE 2004-IT Education-The State of the Art), 169-176.

[6] D. Moore. 2014. Supporting students in music technology higher education to learn computer programming. *Journal of Music, Technology, and Education*, 1(1), 75-92.

[7] S. E. George. 2001. Learning and the reflective journal in computer science. In the *Proceedings of the 25th Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 77-86.

[8] A. Fekete, J. Kay, J. Kingston and K. Wimalaratne. 2000. Supporting reflection in introductory computer science. In *ACM SIGCSE Bulletin*, 32(1), 144-148.

[9] C.J. Van Wyck. 1995. Programming as writing: Using portfolios. In *ACM SIGCSE Bulletin*, 27(4), 39-42.

[10] B.C. Ladd. 2003. It's all writing: Experience using rewriting to learn in introductory computer science. *Journal of Computing Sciences in Colleges*, 18(5), 557-64.

[11] E. Lahtinen, K. Ala-Mutka and H.M. Järvinen. 2005. A study of the difficulties of novice programmers. In *ACM SIGCSE Bulletin*, 37(3), 14-18.

[12] P.M. Papadopoulos, S.N. Demetriadis, I.G. Stamelos, and I.A. Tsoukalas. 2011. The value of writing-to-learn when using question prompts to support web-based learning in ill-structured domains. *Educational Technology Research and Development*, *59*(1), 71-90.

[13] D.E. Knuth. 1984. Literate programming. *The Computer Journal*, *27*(2), 97-111.

[14] A. Strauss and J. Corbin. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Thousand Oaks, SAGE.

[15] D. Steidl, B. Hummel and E. Juergens. 2013. Quality analysis of source code comments. In Proceedings of *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, 83-92.

[16] D.J. Lawrie, H. Feild, and D. Binkley. 2006. Leveraged quality assessment using

information retrieval techniques. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 149-158.

[17] N. Khamis, R. Witte and J. Rilling. 2010. Automatic Quality Assessment of Source Code Comments: The JavadocMiner. In *NLDB*, 68-79.

[18] L. Tan, D. Yuan and Y. Zhou. 2007. Hotcomments: how to make program comments more useful? In *HOTOS '07*.

[19] S. Margaret-Anne, J. Ryall, R.I. Bull, D. Myers and J, Singer 2008. To do or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08,* 251-260.

[20] T.J. McGill and S.E. Volet.1997. A conceptual framework for analyzing students' knowledge of programming. *Journal of research on Computing in Education*, *29*(3), 276-297.