A Case Study of Writing to Learn to Program: Codebook Implementation and Analysis

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 use of digital systems to measure and support engineering education, specifically through learning analytics and the pedagogical uses of digital systems. She also investigates fundamental questions critical to improving undergraduate engineering degree pathways. . 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 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. 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.

A Case Study of Writing to Learn to Program: Codebook Implementation and Analysis

1. Introduction

In this research paper, we explore the application of our qualitative codebook by conducting a comparative investigation of three exemplar introductory programming lab submissions. We selected three lab samples from an introductory programming course in python for the comparison. Each lab submission was chosen to showcase a variety of thinking processes and organizational strategies to assist in illustrating the reasoning behind our qualitative coding methods. The solutions presented come from a single assignment across three different sections in the same semester. This was to make clear the patterns presented and to demonstrate the amount of diversity that can be displayed within the context of a single assignment. It is important to note that we are not analyzing the assignment solutions for correctness but only looking at the thinking and organizational strategies used at this time.

2. Theoretical Foundations based in Writing to Learn

Learning to program is a complex process that could benefit from Writing to Learn (WTL) strategies. The struggles of novice programmers is well documented [1]. A commonly cited factor is "fragile knowledge," which is knowledge that is incomplete and superficial [2]. 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]. WTL strategies can promote deeper understanding in any discipline. By making thinking and organization visible, these short, low-stakes writing activities support metacognition. Through reflecting on their thinking processes, learners can recognize what and how they are learning [4]. In the programming process, writing intermingled with coding allows students to reflect in real-time about the choices they are making and the reasons for those choices [5,6]. Just as importantly, examining novice programmers' source code comments can provide insight into their thinking processes as they approach a lab assignment.

3. Conceptual Framework

To investigate intermingled writing and coding, we previously modified lab assignment instructions in some sections (i.e., WTL sections) of an introductory programming course and used the traditional, unmodified assignment in the remaining sections (i.e., TRAD sections). The modified assignments incorporated WTL strategies and placed an emphasis on writing during the programming process. We used programming submissions from both the WTL and TRAD sections to develop a qualitative code book for analyzing students' Thinking Processes and Visual Organization Strategies. We briefly define Thinking Processes and Visual Organization

Strategies in this section, but refer the reader to our previous publications [7,8] for additional details.

3.1 Overview of Thinking Processes

One of our main objectives with WTL is to improve students' problem-solving capacities by moving their focus from procedural knowledge to conceptual and strategic knowledge [9]. Thinking Processes reflect a student's level of strategic knowledge and metacognition. We developed Thinking Processes to give us a method for analyzing the level of reasoning at which students are currently performing when designing and writing a programming solution. Our initial analysis of 172 students' lab assignments has resulted in the development of five major categories of Thinking Processes: *literal, conceptual, reflective, organizational, insufficient and none.* A brief definition of each class is outlined in *Table 1.*

Thinking Processes : Comment Classification	
Organizational	Indicates the beginning of a new section of code with separate functionality, demonstrating an attempt to organize code in the way that they perceive it.
Reflective	Representative of internal dialogue, explains ***why*** a particular approach was taken.
Conceptual	Explains ***how*** the code works or ***what*** it does without restating the code in English, adds additional understanding and improves readability and comprehension of the program from an outside perspective.
Literal	Restates the source code or calculation in English and does not contribute to the understanding of source code.
Insufficient	Comment is too short to warrant a more complex classification, or the comment adds no additional value.
None	Source code was submitted without comments

Table 1: Thinking Processes

Visual Organization : Structure Classification	
Every-line	Characterized by comments that precede or sit side-by-side nearly every line of code.
Unitization	The code is visually organized into units where a **unit** is defined as a section of comments followed by code that is related in functionality or action.
Block-level	Code is visually organized into blocks where comments are preceded by large blocks of code Usually this code could benefit from being broken down into smaller units of organization.
Insufficient	There are not enough comments in the source code to determine a meaningful organizational classification.
None	No clear visual organization strategy.

Table 2: Visual Organization

3.2 Overview of Organizational Strategies

Organizational Strategies allow us to answer the question of how a student utilizes their writing and comments to visually structure their code. We take into consideration various characteristics such as the use of white-space, commenting patterns and the size of continuous code segments, which we refer to as **units**, to gain an understanding of how students visually communicate their ideas and design process through the structure of their code. Large units that contain multiple ideas and could benefit from being broken down into smaller units of organization are **blocks**. Organizational strategies are broken down into five classes: *Every-line, Unitization, Block-level, Insufficient and None.* We have outlined our definitions of these classes in *Table 2.* To see more examples and to read a more detailed explanation of thinking processes and visual organization we refer you to our previous work [8].

4. Case Study: Three Programming Assignment Submissions

The lab assignment we selected for analysis was a two-week pair programmed lab assignment where each group comprised two students. Students were given a set of images and a module that contained an image comparison algorithm and a set of functions to assist them with the task. The goal of the assignment was for the code to prompt the user for the image that best represents the person they are looking for and then to conduct a search that compares the selected image with all other images and return the best match. This was the sixth lab out of ten for the semester, and at this time the students were not expected to utilize functions. The overall programming objectives for this lab were to practice repetition structures, branching, and indexing into lists. For purposes of analysis, all header information has been removed from the Traditional labs, and all header info, test cases, and discussion questions have been removed from the WTL labs in order to focus solely on the source code.

To analyze and demonstrate the application of our codebook, we examine three cases, Case A: *Block-level*, Case B: *Unitization*, and Case C: *Every-line*. We used the visual organization classification to distinguish between each case because, currently, each lab submission can only be assigned a single class within this category. We have chosen not to include *Insufficient* or *None*, as those visual organization classifications are determined by the absence of writing and are self-evident in terms of analysis.

We classify every lab submission in two phases. In the first phase we determine Visual Organization strategy. First, we identify all the comments in a file. Then, we examine the visual pattern created by the placement of comments within the source code to determine a visual organization strategy. As part of previous research efforts [10] a set of features, which is summarized in *Table 3*, was developed to help support Visual Organization classification. Each visual organization strategy has a set of distinct properties that is illustrated with these features and support a given classification. Specifically, we use the Number of Units in a file and the Ratio of Comments to Code to provide strong supporting evidence. To see an overview of the ratio of comments to code for each lab, see *Fig. 1.* In the second classification phase, we consider each comment in conjunction with the source code that follows it in order to assign a classification. We can see a comparison of the number and type of comment classifications made for each lab submission in *Fig. 2.* In the remainder of this section, we examine our classification techniques in more detail.

Summary of Lab Features			
Lab	WL6_S5_G6	TL6_S7_G6	TL6_S9_G5
Organization Strategy	Every-line	Unitization	Block-level
Total Lines in File	42	29	27
Number of Comments Lines	19 (45.23%)	9 (31.03%)	4 (14.81%)
Number of Code Lines	23 (54.75%)	20 (68.97%)	23 (85.19%)
Ratio of Comments to Code	0.8261	0.45	0.17
Number of Units	19	9	3
Ratio of Units to Lines	0.45	0.31	0.111

Table 3: Summary of Lab Features

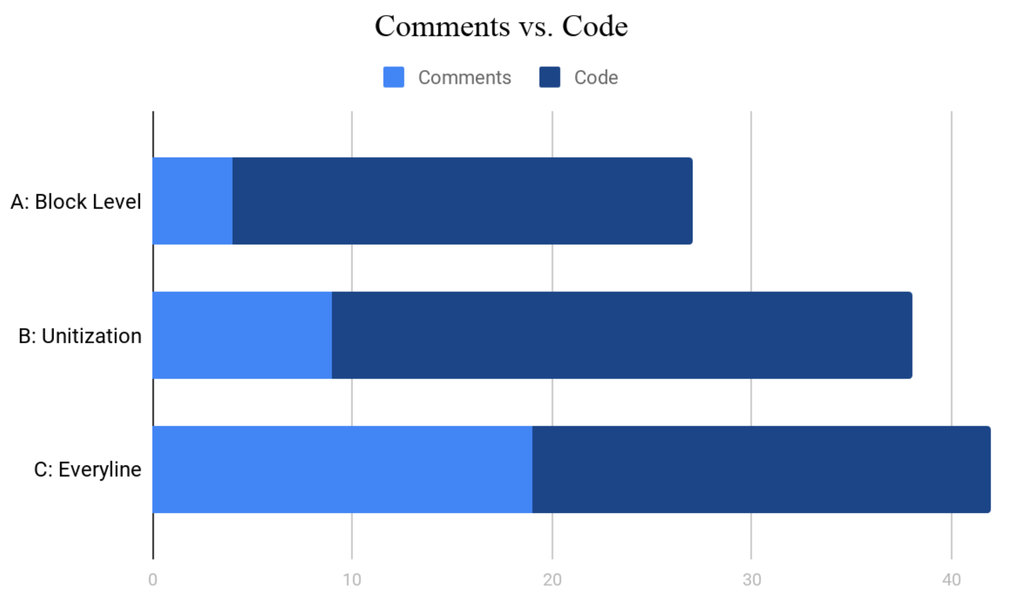


Figure 1: Illustration of the relationship between comments and code for each lab sample.

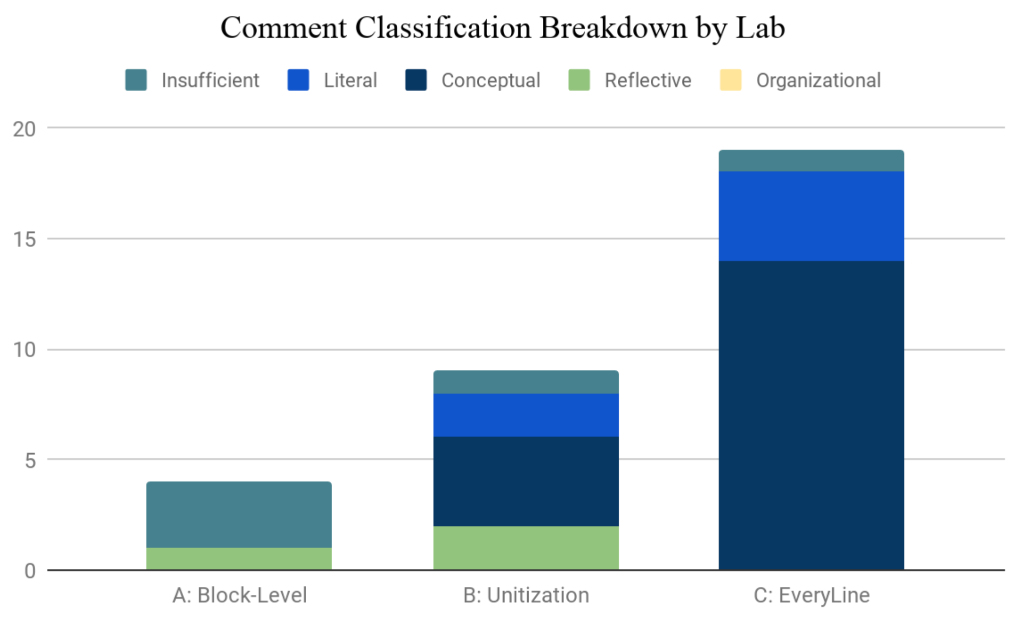


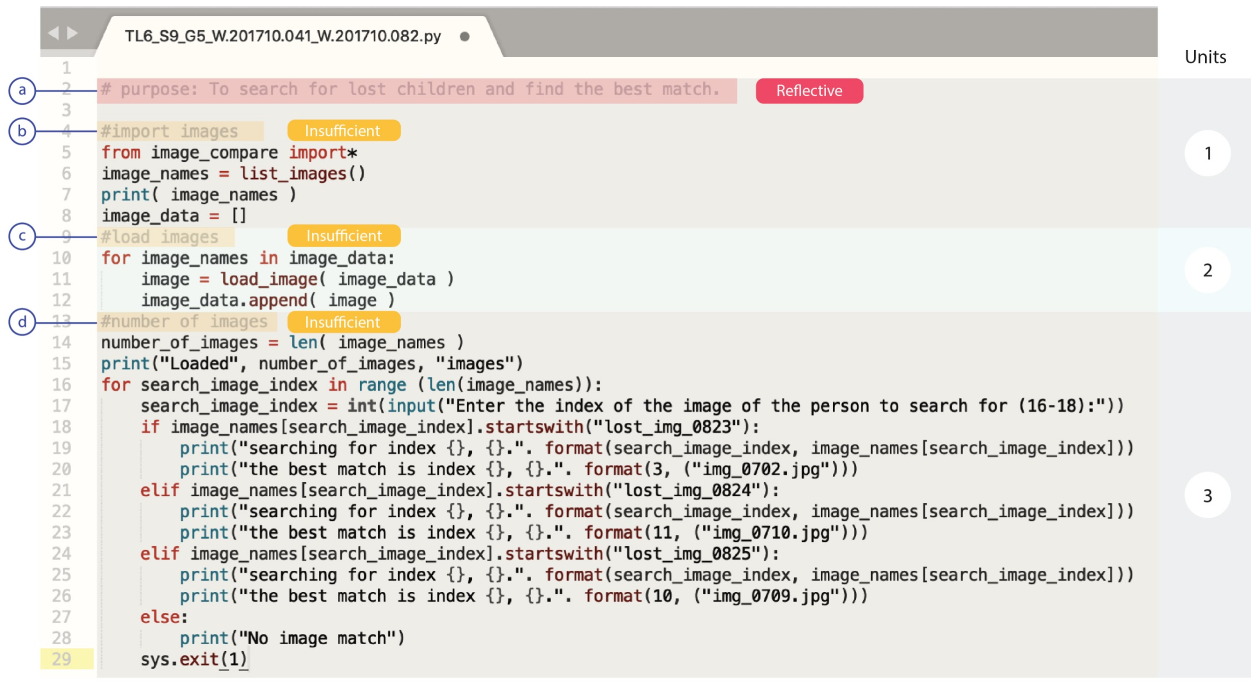
Figure 2: Visualization of the number and type of comments in each sample

4.2 Case A: Block-level

For Case A we will examine a lab that has been classified as *Block-Level.* As seen in *Fig. 3,* each comment has been highlighted and classified with a color-coded tag and assigned a label on the left for ease of reference. Each unit has been identified and assigned a number on the right.

First, we will analyze the Organizational Strategy. An initial pass through the source code of lab TL6_S9_G5 tells us that comments and whitespace have been used sparingly and we can identify large units of code that follow a 1-2 line comment. Analyzing each of these units individually, we observe that all the source code in one unit does not directly relate to the comment. For example, unit 2 line 12 contains an "append" command, which is not clearly related to the comment in line 9 "#load images". Thus, these units are blocks. With regards to our metrics, there is a low comment to code ratio of 0.17 and the lab consists of 14.81% comments and 85.19% code. This submission is the most concise solution of the three cases at 27 lines, not including white-space. When comparing the number of units to the total lines in the file we get a very low ratio of 0.11. Consistent with *Block-level* properties, we observe a low comment to code ratio, a small number of units, and a low unit to total lines ratio. All of these metrics are much smaller in comparison to *Unitization* or *Every-line* strategies resulting in a firm classification of *Block-level* organization.

Next, we consider Thinking Processes by analyzing each source code comment. The classification of comments resulted in 1- *reflective and* 3 - *insufficient* as illustrated in *Fig. 3*. Comment ***a*** is *reflective,* while fairly short for a reflective comment, it provides the reason for



```python

# purpose: To search for lost children and find the best match.

#import images
from image_compare import*
image_names = list_images()
print( image_names )
image_data = []
#load images
for image_names in image_data:
    image = load_image( image_data )
    image_data.append( image )
#number of images
number_of_images = len( image_names )
print("Loaded", number_of_images, "images")
for search_image_index in range (len(image_names)):
    search_image_index = int(input("Enter the index of the image of the person to search for (16-18):"))
    if image_names[search_image_index].startswith("lost_img_0823"):
        print("searching for index {}, {}.". format(search_image_index, image_names[search_image_index]))
        print("the best match is index {}, {}.". format(3, ("img_0702.jpg")))
    elif image_names[search_image_index].startswith("lost_img_0824"):
        print("searching for index {}, {}.". format(search_image_index, image_names[search_image_index]))
        print("the best match is index {}, {}.". format(11, ("img_0710.jpg")))
    elif image_names[search_image_index].startswith("lost_img_0825"):
        print("searching for index {}, {}.". format(search_image_index, image_names[search_image_index]))
        print("the best match is index {}, {}.". format(10, ("img_0709.jpg")))
    else:
        print("No image match")
    sys.exit(1)
```

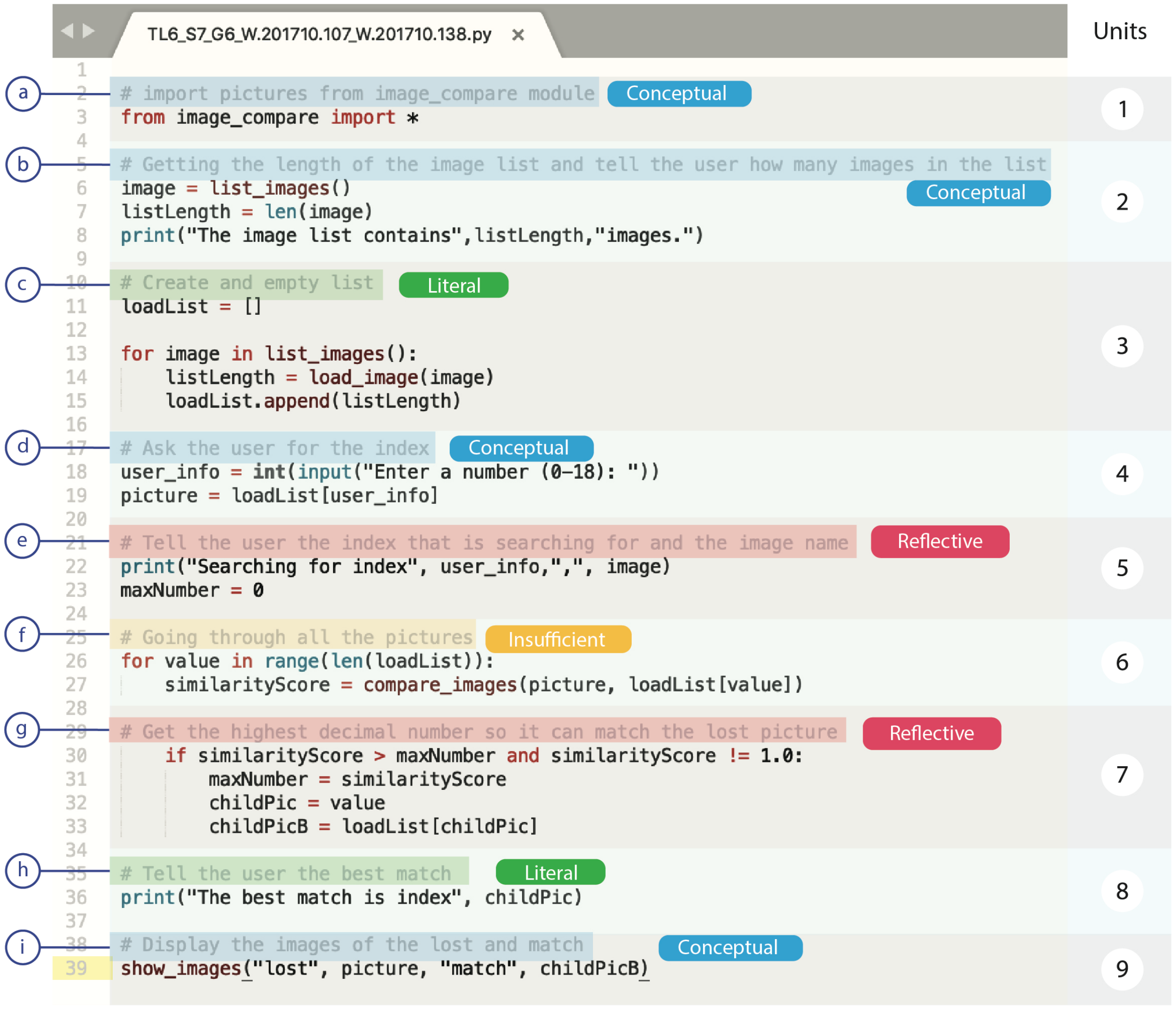
Figure 3: Case Study A: Block-level. Source code for Lab Sample TL6_S9_G5.

and states the overall goal of the program providing insight into the function of the code. It illustrates the question of why without providing implementation details. Comments ***b, c,*** and ***d*** are all classified as *insufficient*. Each of these comments on their own adds little value to the understanding of the code and are too short to be assigned a more specific classification. It is possible that comments **b** and ***c*** could be considered *organizational*, but due to the length and a lack of consistency in the organizational strategy we lack sufficient evidence to warrant a more complex classification.

4.3 Case B: Unitization

For Case B be we will illustrate the properties of *Unitization* combined with a variety of commenting types. *Fig. 4* provides a visual overview of how we classified each comment of lab TL6_S7_G6 and how the code breaks down into individual units. When analyzing the Organizational Strategy, an initial pass through the code conveys a strong sense of clear consistent commenting and use of white space to organize code into logical groupings that form easily distinguishable units. We have a nine distinct units and a total of 29 lines, not including white-space where each unit consists of a single comment followed by one or more source code statements where the minimum is one line and the maximum is four lines of code. The ratio of comments to code is closer to 2:1 at 0.45 with 31.03% comments and 68.97% code. Comparing the number of units to the total number of lines results in a value of 0.31. Here we can see that we have more units and more comments than in the *Block-level* strategy and fewer units and fewer overall comments than that of an *Every-line* organizational strategy, which we will look at in the third case. The units are larger than *Every-line* units but too small to be considered *Block-level* units because most units would not benefit from being broken down into smaller units. That is to say, it would not increase the clarity or organizational structure of the program. Combined these properties lead us to a clear classification of *Unitization*.

To analyze Thinking Processes we have classified each of the comments in the source code resulting in 4 - *Conceptual*, 2 - *Reflective*, 2 - *Literal* and 1 - *Insufficient* as seen in *Fig. 4*. Comments **a, b, d**, and **i** are *Conceptual*. When deciding if a comment is *conceptual*, a clear way



```

# import pictures from image_compare module
from image_compare import *

# Getting the length of the image list and tell the user how many images in the list
image = list_images()
listLength = len(image)
print("The image list contains",listLength,"images.")

# Create and empty list
loadList = []

for image in list_images():
    listLength = load_image(image)
    loadList.append(listLength)

# Ask the user for the index
user_info = int(input("Enter a number (0-18): "))
picture = loadList[user_info]

# Tell the user the index that is searching for and the image name
print("Searching for index", user_info,",", image)
maxNumber = 0

# Going through all the pictures
for value in range(len(loadList)):
    similarityScore = compare_images(picture, loadList[value])

# Get the highest decimal number so it can match the lost picture
    if similarityScore > maxNumber and similarityScore != 1.0:
        maxNumber = similarityScore
        childPic = value
        childPicB = loadList[childPic]

# Tell the user the best match
print("The best match is index", childPic)

# Display the images of the lost and match
show_images("lost", picture, "match", childPicB)
```

Figure 4: Case Study B: Unitization. Source code for Lab Sample TL6_S7_G6.

to do so is to ask if it answers *How?* or *What?* (e.g., “If the student states they are making a variable or calling a function, does it say what the purpose is?”). For example, if we look at **a,** it tells us what they need to import the image_compare module for importing pictures. In comment **b,** they give us a summary of what they intend to do in the next few lines of code. They don’t reflect on what the list_images() function does and the only information we can gather is that it is a list but it is enough information to prevent it from being a *literal* comment. We don't know what form the image data is in. We also don't know the purpose of calculating the length and how it will be used later in the program. Comment **b**, like many *conceptual* comments, is immediate, reflecting only on this single moment in the code and does not provide details on how the code relates to previous or future decisions. Comment ***d*** tells us what they are asking the user for, it provides more information than what is in the initial input prompt by stating that it is an index but they do not tell us why they need the index from the user. If they did then we could consider this to be *reflective*. The last *conceptual* comment is **i** which helps us understand what the show_images function does. The function is self-documenting but we want to maintain a naive point of view and refrain from making any assumption about what the code does. From this comment, we can gather that they will display the initial image they were matching against and the resulting match that was returned from their search.

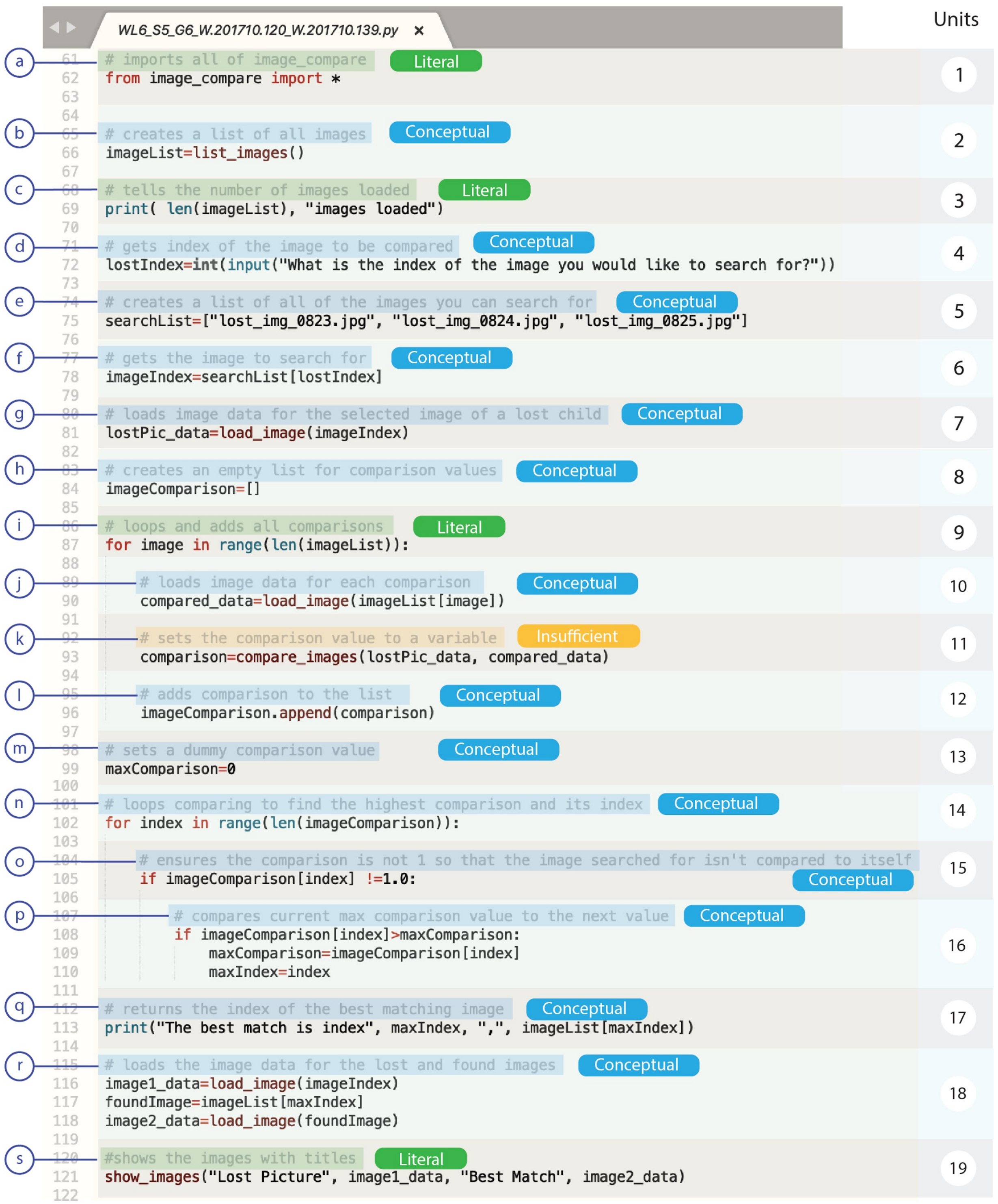
Comments **e** and **g** are *reflective*. When analyzing reflective comments it can be challenging to determine when a comment goes from being *conceptual* to *reflective*, often it is the difference between a few keywords. There are also times when the first part of a comment will be *conceptual* or *literal* but they add additional commentary to the end that indicates their internal reasoning. If we look at comment **e,** the programmer is trying to tell us why they need to print the index and that this is the name and the index of the image that is being searched for, helping us to understand why they need to output the index. This is in contrast to the above comment **d** if **e** was only 'tell the user the index' it would have been *conceptual* as it would tell us what is being done but not why. Comment **g** has a similar structure where the first half "Get the highest decimal number" tells us what is going to happen in the code and the second half "so it can get the lost picture" tells us why they need to. Often when we see 'so' as a conjunction in a comment it is a strong indicator that they are going to give a reason why shifting the comment from *conceptual* to *reflective*.

Finally, we have the *literal* comments **c** and **h** and the *insufficient* comment **f.** Comment **c** provides little information and does not tell us why or what the list will be for. It can be seen as an English restatement of creating an empty list in python. Comment **h** simply parrots the print statement below it and can be seen as unnecessary as it provides no additional insight. Comment **f** is insufficient because it has a vague and casual tone and provides little to no clarity on the purpose of the code below it. If **f** was "looping through all the pictures" it would be *literal.* If **f** was "searching through all the pictures, it could be considered *conceptual.* However, since the provided comment uses language that is not specific enough to warrant a more complex classification, the result is an *insufficient* classification.

4.4 Case C: Every-line

For Case C we will analyze the properties of an *Every-line* Organizational Strategy. *Fig. 5* provides a visual overview of how we classified each comment of lab WL6_S5_G6 and how the code breaks down into individual units. Upon initial inspection, we observe that there is a consistent pattern where, for a majority of units, a single line of code follows a single comment. Lab WL6_S5_G6 exhibits strong *Every-line* Characteristics in which the number of lines of code and the number of lines of comments are balanced and exhibit a code to comment ratio of .82 with comments preceding 19 out of 23 lines of code. This is consistent with our prior work indicating that the ratio of comments to code in an *Every-line* style organization is close to 1. Additionally, there is little to no separation of code into larger logical units; equal spacing is used throughout the program. The only grouping of concepts comes in the form of programming structures such as for-loops and if-statements. There are no comments indicating organization, and all comments are short single-line comments. All these metrics are consistent with an *Every-line* visual organization classification.

Considering thinking processes, the classification of comments resulted in 14 - *conceptual*, 4 - *literal* and 1- *insufficient*. The *conceptual* comments are ***b***, ***d*** - ***f, j,*** and ***l*** - ***r***. These comments all explain the function of the code without simply restating the code. For example, comment ***n*** uses the word "loops", an indication that it could be a literal restatement of code, but the comment continues to explain that the source code is "comparing to find the highest comparison and its index", which provides additional insight into the source code functionality that is not simply a restatement.



```python
# imports all of image_compare
from image_compare import *


# creates a list of all images
imageList=list_images()

# tells the number of images loaded
print( len(imageList), "images loaded")

# gets index of the image to be compared
lostIndex=int(input("What is the index of the image you would like to search for?"))

# creates a list of all of the images you can search for
searchList=["lost_img_0823.jpg", "lost_img_0824.jpg", "lost_img_0825.jpg"]

# gets the image to search for
imageIndex=searchList[lostIndex]

# loads image data for the selected image of a lost child
lostPic_data=load_image(imageIndex)

# creates an empty list for comparison values
imageComparison=[]

# loops and adds all comparisons
for image in range(len(imageList)):

    # loads image data for each comparison
    compared_data=load_image(imageList[image])

    # sets the comparison value to a variable
    comparison=compare_images(lostPic_data, compared_data)

    # adds comparison to the list
    imageComparison.append(comparison)

# sets a dummy comparison value
maxComparison=0

# loops comparing to find the highest comparison and its index
for index in range(len(imageComparison)):

    # ensures the comparison is not 1 so that the image searched for isn't compared to itself
    if imageComparison[index] !=1.0:

        # compares current max comparison value to the next value
        if imageComparison[index]>maxComparison:
            maxComparison=imageComparison[index]
            maxIndex=index

# returns the index of the best matching image
print("The best match is index", maxIndex, ",", imageList[maxIndex])

# loads the image data for the lost and found images
image1_data=load_image(imageIndex)
foundImage=imageList[maxIndex]
image2_data=load_image(foundImage)

#shows the images with titles
show_images("Lost Picture", image1_data, "Best Match", image2_data)
```

Figure 5: Case Study C: Every-line. Source code for Lab Sample WL6_S5_G6.

The *literal* comments are ***a***, ***c***, ***i*** and ***s***. **a** and **c** are a plain English restatement, and **i** borders between *literal* and *conceptual* as the comment has two parts. First, they state that they are ‘looping’ which is *literal* and then they state ‘adds all comparisons’, which could be *conceptual,* except that it does not add to the understanding of the code. Instead the statement creates more questions for the reviewer (e.g., *Are we adding values or appending items?* and *What are we comparing?*). By reviewing the code, we can see they are appending the results from the comparison for each image to a list, but this is not clear from the comment. The final *literal* comment ***s*** is a restatement of the function name and provides no additional description. In a way, the comment is unwarranted as the function itself is self-documenting, but we can see that the student still felt they needed to place a comment above it, which is consistent with the *Every-line* pattern that is often characterized by over commenting.

Line ***k*** is classified as *insufficient* because the comment provides no meaningful information and is almost nonsensical, which could indicate the student's misunderstanding of what is actually happening in the code. While most *insufficient* comments tend to be those that are too short, there are cases of longer comments that still do not provide any meaningful information.

4. Conclusions and Future Work

In this paper, we have demonstrated the application of our qualitative codebook and how it can be utilized to analyze introductory students' source code and comments. Our case study looked at three source code samples from an introductory programming lab assignment where we identified and analyzed a diverse set of patterns that illustrate the Thinking Processes and Organizational Strategies of introductory programming students. This case study demonstrates that student programming assignments contain distinct patterns with recognizable characteristics that can be identified and used to understand as students' metacognition and design processes. Being able to identify the patterns displayed by successful students versus struggling students will 1) give us a toolset to better guide students in how to self-assess and monitor their own learning progress, and 2) help educators understand how writing and visual organization impacts a student's thinking, problem solving, and design process.

Our qualitative codebook contains definitions and examples to assist our qualitative coding team with determining the appropriate classification. When analyzing source code writing it may not always be clear what the correct classification should be. We have included an "unable to classify category" of *Insufficient* for those cases. The validity of our work is strengthened by ensuring that there is clear consistent reasoning for assigning a given classification.

Now that the codebook has been formalized and we are beginning the process of coding our entire dataset, we anticipate there will be modifications and clarifications to the codebook. Further classification categories may be warranted and we are open to starting a discourse on this topic. Additionally, we are aware that the classification of Visual Organization is restricted to a single classification but portions of the code may exhibit multiple organizational characteristics. We are developing methods for analyzing units on a more granular level that allow for multiple organizational styles to exist within a single file. While the labs presented in this paper do have a single organizational strategy, there are a variety of cases such as *Unitization* being used as the primary organizational structure but demonstrate *Every-line* strategies inside various programming structures such as functions. We are investigating ways to indicate these sub-organizational strategies on a unit by unit basis.

Acknowledgement
This material is based upon work supported by the National Science Foundation under Grant No. DUE-1612132. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Watson, Christopher, and Frederick WB Li. "Failure rates in introductory programming revisited." *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 2014.

[2] Perkins, David, and Fay Martin. "Fragile Knowledge and Neglected Strategies in Novice Programmers. IR85-22." (1985).

[3] Bergin, Susan, Ronan Reilly, and Desmond Traynor. "Examining the role of self-regulated learning on introductory programming performance." *Proceedings of the first international workshop on Computing education research*. ACM, 2005.

[4] Emig, Janet. "Writing as a mode of learning." *College composition and communication* 28.2 (1977): 122-128.

[5] Knuth, Donald Ervin. "Literate programming." *The Computer Journal* 27.2 (1984): 97-111.

[6] Jones, B.A., Mohammadi-Aragh, M.J., Barton, A.K., Reese, D., & Pan, H. (2015). Writing-to-Learn-to-Program: Examining the need for a new genre in programming pedagogy. 122nd ASEE Annual Conference and Exposition, Seattle, Washington.

[7] Mohammadi-Aragh, M.J., Beck, P.J., Barton, A.K., Reese, D., Jones, B.A., Jankun-Kelly, M (2018). Coding the coders: A qualitative investigation of students' commenting patterns. American Society for Engineering Education Annual Conference and Exposition.

[8] Mohammadi-Aragh, M. J., Beck, P. J., Barton, A. K., Reese, D. S., Jones, B. A., Jankun-Kelly, M. (2018). Coding the Coders: Creating a Qualitative Codebook for Students' Commenting Patterns. SIGCSE '18: The 49th ACM Technical Symposium on Computing Science Education Proceedings. Baltimore, MD.

[9] McGill, Tanya J., and Simone E. Volet. "A conceptual framework for analyzing students' knowledge of programming." *Journal of research on Computing in Education* 29.3 (1997): 276-297.

[10] Beck, P.J., Mohammadi-Aragh, M.J., Archibald, C., Jones, B.A., and Barton, A. (2018). Real-time Metacognitive Feedback for Introductory Programming Using Machine Learning. IEEE Frontiers in Education (FIE).