CODE DECOMPOSITION: A NEW HOPE

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Nupur Garg

June 2017

COMMITTEE MEMBERSHIP

TITLE:	Code Decomposition: A New Hope
AUTHOR:	Nupur Garg
DATE SUBMITTED:	June 2017
COMMITTEE CHAIR:	Aaron Keen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Zoë Wood, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Theresa Anne Migler-VonDollen, Ph.D. Lecturer of Computer Science

ABSTRACT

Code Decomposition: A New Hope

Nupur Garg

Code decomposition (also known as functional decomposition) is the process of breaking a larger problem into smaller subproblems so that each function implements only a single task. Although code decomposition is integral to computer science, it is often overlooked in introductory computer science education due to the challenges of teaching it given limited resources.

Earthworm is a tool that generates unique suggestions on how to improve the decomposition of provided Python source code. Given a program as input, *Earthworm* presents the user with a list of suggestions to improve the functional decomposition of the program. Each suggestion includes the lines of code that can be refactored into a new function, the arguments that must be passed to this function and the variables returned from the function. The tool is intended to be used in introductory computer science courses to help students learn more about decomposition.

Earthworm generates suggestions by converting Python source code into a control flow graph. Static analysis is performed on the control flow graph to direct the generation of suggestions based on code slices.

ACKNOWLEDGMENTS

Thanks to:

- my mother and father, for support me in every way
- my sister, for her support and her colorful pens to help me edit my thesis
- Dr. Aaron Keen, for being an incredible thesis advisor and professor
- Dr. Zoë Wood, for being an amazing mentor and professor
- Dr. Theresa Migler-VonDollen, for being on my committee
- Amy Tsai, for sending me pictures of minions and always encouraging me
- Nick Gonella, for helping me TA and grade for my class
- Auberon Lopez, for always sending me positive messages
- Andrew Wang, for helping keep me on track with our weekly check ins
- Max Zinkus, for editing my paper
- Daniel Kauffman, for editing my survey and understanding the challenges of being a student-teacher trying to defend
- Mauri Laitinen, my high school robotics advisor who taught me to love learning
- Andrew Guenther, for uploading this template
- the department, for always supporting my endeavors
- all my family and friends who I did not list here, for always supporting me

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

Chapter 1

INTRODUCTION

Computing has become central to life in the 21st century. With the prominence of technology from our personal computers to our pasttime activities to our methods of transportation, jobs in computing are becoming increasingly available. According to the U.S. Bureau of Labor Statistics, there will be a 17% growth in software developer jobs by 2024[7]. With students from increasingly diverse backgrounds being encouraged to pursue a computer science degree, it is critical to begin exploring additional methods to teach the same concepts to cater to the various learning styles and background knowledge of these students [8, 12].

At its heart, computer science is problem solving. One of the important skills to learn and practice early is the process of breaking down a complex problem into simpler subproblems that can be used to solve the overall problem - a process known as functional decomposition.

In primary and secondary education in America, few subjects focus on problem solving and even fewer emphasize this process of solving a problem by breaking it into subproblems. As a tutor for introductory computer science courses for the last four years, I have found many students in introductory courses have a difficult time understanding how to decompose a problem, in part due to a lack of exposure to this process.

Code decomposition is important because it helps us improve testability, reusability, and readability - all key components to a well-engineered software system. The code in Listing 1.1 and in Listing 1.3 perform identical tasks. Listing 1.1 is adapted from student code while Listing 1.3 is an improved version of the initial listing. Both

are part of a function within a word search program that finds words arranged vertically in the puzzle. However, the task is more apparent in Listing 1.3 by virtue of the decomposition based on suggestions provided in Listing 1.2. A bug would be easier to isolate using unit tests for the code in Listing 1.3. Decomposition is an important concept to teach early to make it a central part of students' coding skills instead of an afterthought.

Listing 1.1: Poor code decomposition

```
49 def check_cols(puzzle, word):
50     newword = []
51     j = len(word)
52     while j > 0:
53         newword.append(word[j-1])
54         j -= 1
55     backWord = ''.join(newword)
56     newpuzzle = []
57     for col in range(len(puzzle)):
58         new = []
59         for row in range(len(puzzle)):
60             new.append(puzzle[row][col])
61         newpuzz = ''.join(new)
62         newpuzzle.append(newpuzz)
63     for col in range(len(newpuzzle)):
64         if word in newpuzzle[col]:
65             length = 0
66             row = 0
67             for char in newpuzzle[col]:
68                 letter = word[length]
69                 if char != letter:
70                     length = 0
71                     letter = word[length]
72                 if char == letter:
73                     length += 1
74                     if len(word) == length:
75                         row -= (length-1)
76                         place = ['(DOWN)', row, col]
77                         return place
78             row += 1
```

Listing 1.2: Suggestions generated by Earthworm to improve Listing 1.1

```
line 63-78 (check_cols):
        parameters: newpuzzle, word
        returns: place
        reason: Removing these instructions decreases number
                of paths of execution in this function -
                making the code more readable and testable.

line 68-71 (check_cols):
        parameters: char, length, word
        returns: length, letter
        reason: Multiple variables defined prior to these
                instructions are not used in these line
                numbers.

line 72-77 (check_cols):
        parameters: char, col, length, letter, row, word
        returns: length, place, row
        reason: Removing these instructions decreases number
                of paths of execution in this function -
                making the code more readable and testable.
```

Listing 1.3: Improved code decomposition after using Earthworm

```
# Checks if the given letter is equal.
def check_letter(char, length, word):
    letter = word[length]
    if char != letter:
        length = 0
        letter = word[length]
    return (length, letter)

# Checks if a word is found going down in a given column.
def check_down(char, col, length, letter, row, word):
    if char == letter:
        length += 1
        if len(word) == length:
            row -= (length-1)
            place = ['(DOWN)', row, col]
            return length, place, row
    return length, None, row

# Finds a word going down in all columns.
def find_down(newpuzzle, word):
    for col in range(len(newpuzzle)):
        if word in newpuzzle[col]:
            length = 0
            row = 0
            for char in newpuzzle[col]:
```

```
74                  length, letter = check_letter(
75                          char, length, word)
76                  length, place, row = check_down(
77                          char, col, length, letter, row, word)
78                  row += 1
79      return None
80
81  # Searches the columns of the puzzle.
82  def check_cols(puzzle, word):
83      newword = []
84      j = len(word)
85      while j > 0:
86          newword.append(word[j-1])
87          j -= 1
88      backWord = ''.join(newword)
89      newpuzzle = []
90      for col in range(len(puzzle)):
91          new = []
92          for row in range(len(puzzle)):
93              new.append(puzzle[row][col])
94          newpuzz = ''.join(new)
95          newpuzzle.append(newpuzz)
96      return find_down(newpuzzle, word)
```

Despite its importance, many introductory college courses fail to sufficiently teach code composition due to the complications surrounding its teaching. Given students' limited background knowledge, many introductory computer science courses place emphasis on teaching basic coding concepts such as conditionals, loops, and general program flow. With increasing class sizes, it is not uncommon for students to never receive feedback on their functional decomposition. From personal experience as a student and tutor, I have noted most students receive a score on their code based exclusively on some measure of execution correctness such as test cases.

The primary contribution of this work is *Earthworm*, a tool for students that makes suggestions on how to improve the decomposition of their code. In addition, this work outlines the algorithms that help generate these suggestions. The tool is syntax-driven and assumes the user has already developed a syntactically correct solution to the problem. *Earthworm* provides value by giving immediate feedback to

students on how to improve their decomposition. Because *Earthworm* is intended to teach code decomposition to introductory students, the program focuses on identifying code to be restructured. It does not address the root of the problem in that it does not teach decomposition before the student begins coding. However, it is an alternative approach to teaching code decomposition that does not require extensive background knowledge from students and that imposes little burden on instructors, making it seamless to integrate within existing introductory computer science courses.

Earthworm operates on programs written in Python and has been tested on Python 2.7.10 and Python 3.5.2. It uses a variety of algorithms, including code slicing and live variable analysis which are discussed in greater detail in Chapter 2, to provide decomposition suggestions. The application of these algorithms to generate suggestions is outlined in Chapter 4. Each generated suggestion details the methodology behind generating the suggestion to help students learn common patterns they can identify to become self-sufficient in decomposing code.

Although there are no existing tools that perform the task that *Earthworm* aims to accomplish, there are many tools that exist that aid a programmer when refactoring code. Chapter 3 outlines the approach some of these tools use.

Chapters 5 and 6 evaluate the quality of the suggestions and the impact the tool has on student code decomposition. Lastly, Chapter 7 analyzes the areas where this project can be developed in the future. This paper discusses the design, implementation and performance of *Earthworm*. The goal of this thesis is to demonstrate the value that a static, syntax-driven suggestion tool such as *Earthworm* can provide in teaching code decomposition.

Chapter 2

BACKGROUND

Earthworm uses multiple approaches to generate suggestions to improve code decomposition. This chapter covers terms and algorithms associated with these approaches. The approaches themselves are outlined in Chapter 4.

2.1 Restructuring Programs

The central goal of *Earthworm* is to generate suggestions for restructuring the provided code that result in better code decomposition. Restructuring programs requires two main steps [14]:

1. *Identify* - Identify the code segment that can be restructured.
2. *Transform* - Refactor this code into its own function.

Identify determines the parts of the program that can be restructured to improve the decomposition of the code. *Transform* is only required when the tool performs the refactoring for the user. Because *Earthworm* is intended to teach code decomposition to introductory students, the program focuses on identifying code to be restructured and leaves the transforming to the user.

2.2 Control Flow Graph

To support function-level analysis, each function in the student's source code is transformed from text into a control flow graph (CFG). The CFG is a directed graph with a source (start node with no predecessors) and a sink (end node with no

successors) [10]. There is a path from the source to every node in the graph and from every node to the sink.

Each node, referred to as a basic block or, more simply, as a block, contains zero or more instructions and is identifiable by a unique label. Each edge in the graph represents a portion of the control flow of the function. An unoptimized CFG can have an empty block containing no instructions that reflects program structure generated during construction. The sink is a successor to any block containing a 'return' statement.

Each instruction within the basic block contains:

- Line number in the original source code
- Instruction type, for special instructions: `return`, `else`, and function headers
- Variables referenced in the instruction
- Variables defined by the instruction
- Indentation level of the instruction in the original source code
- Line number of logical control to the given instruction, as seen in Figure 2.3 in the instruction on line number `4` contained in block `L2`.
- Line numbers of other instructions, in a multiline instruction. `if`, `elif`, `else` statements and `try`, `except`, `finally` statements are grouped together as a multiline statement.

2.2.1 Important Patterns

Prior to demonstrating an example function, this section demonstrates a few less-intuitive details of implementing a control flow graph by examining subgraphs of a control flow graph.

Figure 2.1 demonstrates how list comprehensions and conditionals with only an if statement, as seen in Listing 2.1, are represented in the control flow graph. Walking through the control flow graph, the first block `L3` contains the conditional (a check of the value of `filter_num`) as well as any instructions prior to it (not shown in this example). The CFG depicts the two possible paths the code can take. If `filter_nums` is `True`, the code follows the edge from block `L3` to block `L4`. Within block `L4`, the list comprehension can be seen through the definition and reference of `num` and `numbers`. Additionally, since instruction `6` is the conditional body, it is logically controlled by instruction 5. Block `L4` then follows the edge to block `L5`. If `filter_nums` is `False`, the code follows the path from `L3` directly to `L5`. The CFG emphasizes the paths available, not the exact conditions that cause a particular path to run. Therefore the CFG only tracks the existence of control flow between blocks `L3`, `L4`, and `L5`, not the condition on which each path is taken.

Listing 2.1: Example of list comprehension and if conditional

```
5 if filter_nums:
6     numbers = [num for num in numbers if num > 5]
7 print(numbers)
```

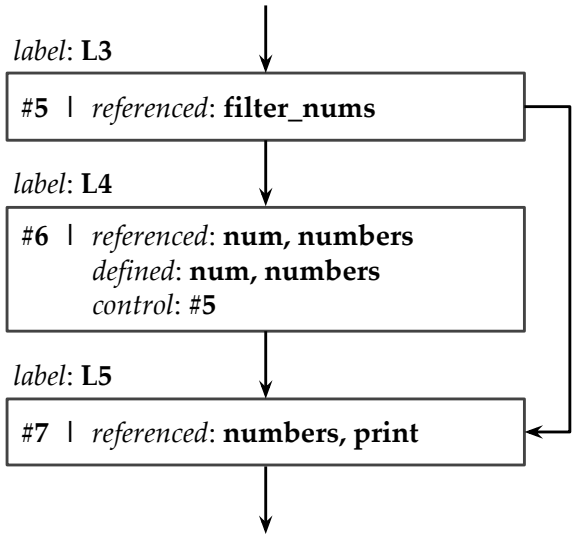

Figure 2.1: Subsection of CFG showing Listing 2.1

Figure 2.2 demonstrates how exception handling in Listing 2.2 translates into a control flow graph. Exception handling is represented less explicitly due to the additional complexity that would be added to the control flow graph if all possible exception paths were fully represented. The first block `L3` contains the `try` command. Following block `L3`, the code within the `try` branch as well as each `except` branch gets its own block that becomes a successor to the current block, `L3`. In an actual `try` block each instruction would have each `except` block as a successor. However, doing so would cause the complexity to increase significantly, as described in 2.6. The program instead focuses on evaluating the complexity of the exception handling block relative to the complexity of other items - such as conditional statements.

The control flow graph ends with the `finally` branch which is treated as straight line code instead of as a branch, since every path has to go through the `finally` block. Any `return` statements within a `try` or `except` block go directly to the sink instead of through the block containing `finally` once again to keep the control flow graph simple. As previously mentioned the `try`, `except`, and `finally` statements are considered as multiline statements to ensure the error catching block will never be split up in a suggestion.

Listing 2.2: Example of exception handling

```
5  try:
6      fp = open(filename, 'r')
7      num = int(fp.read())
8  except FileNotFoundError as e:
9      print('File not found')
10 except ValueError as e:
11     print('Casting failed.')
12 finally:
13     print('Done with program.')
```

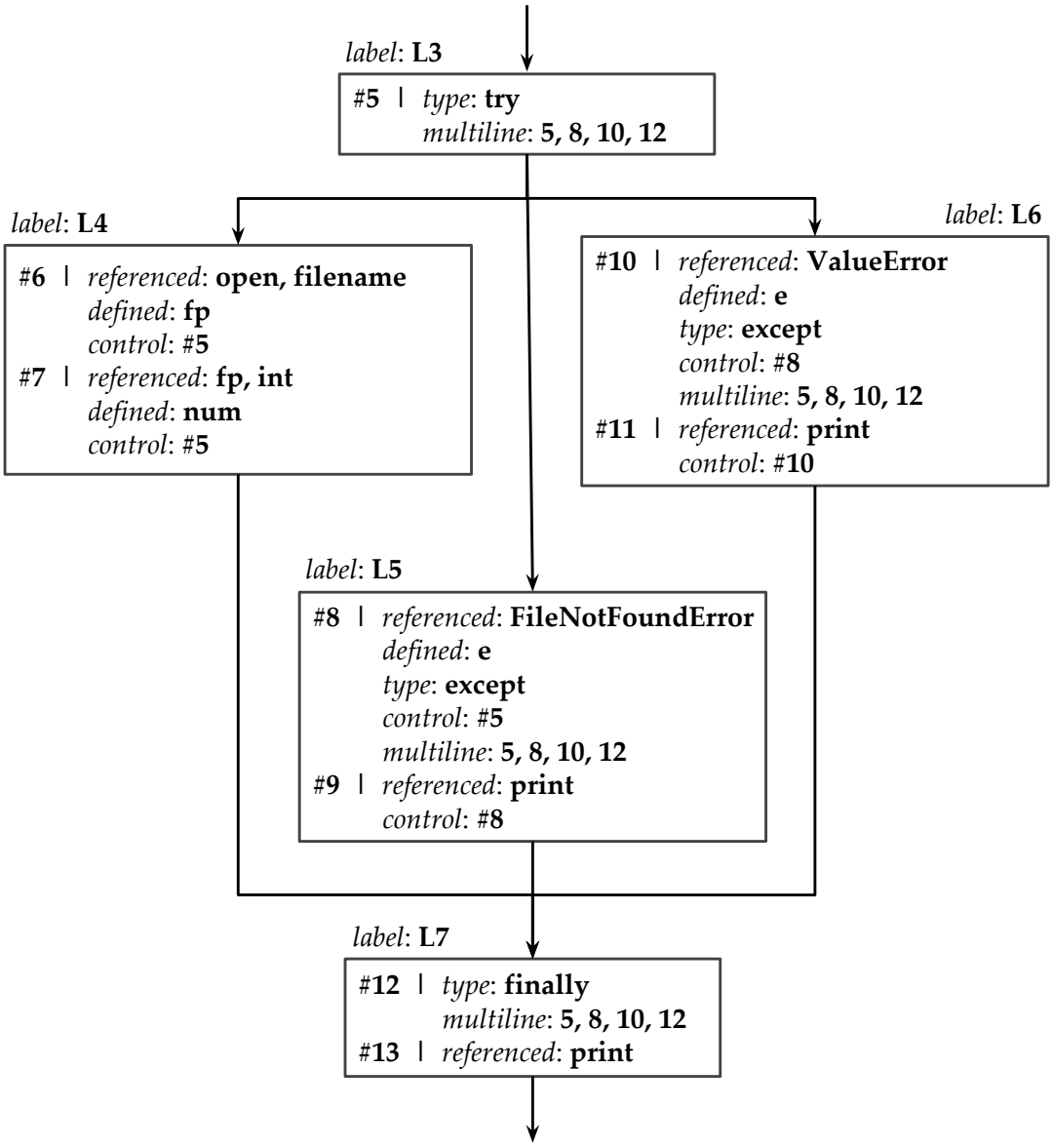

Figure 2.2: Subsection of CFG showing Listing 2.2

2.2.2 Example Functions

Two example functions, one containing a conditional and one containing nested loops, are provided. The CFG for the function in Listing 2.3 containing a conditional is seen in Figure 2.3, and the CFG for the function in Listing 2.4 containing nested `for` loops is seen in Figure 2.4.

Listing 2.3: Conditional example

```
1 # Input: late_days is number of days the rental is late.
2 def rental_late_fee(late_days):
3     if late_days <= 0:
4         return 0
5     elif late_days <= 10:
6         return 5
7     else:
8         return 20
```

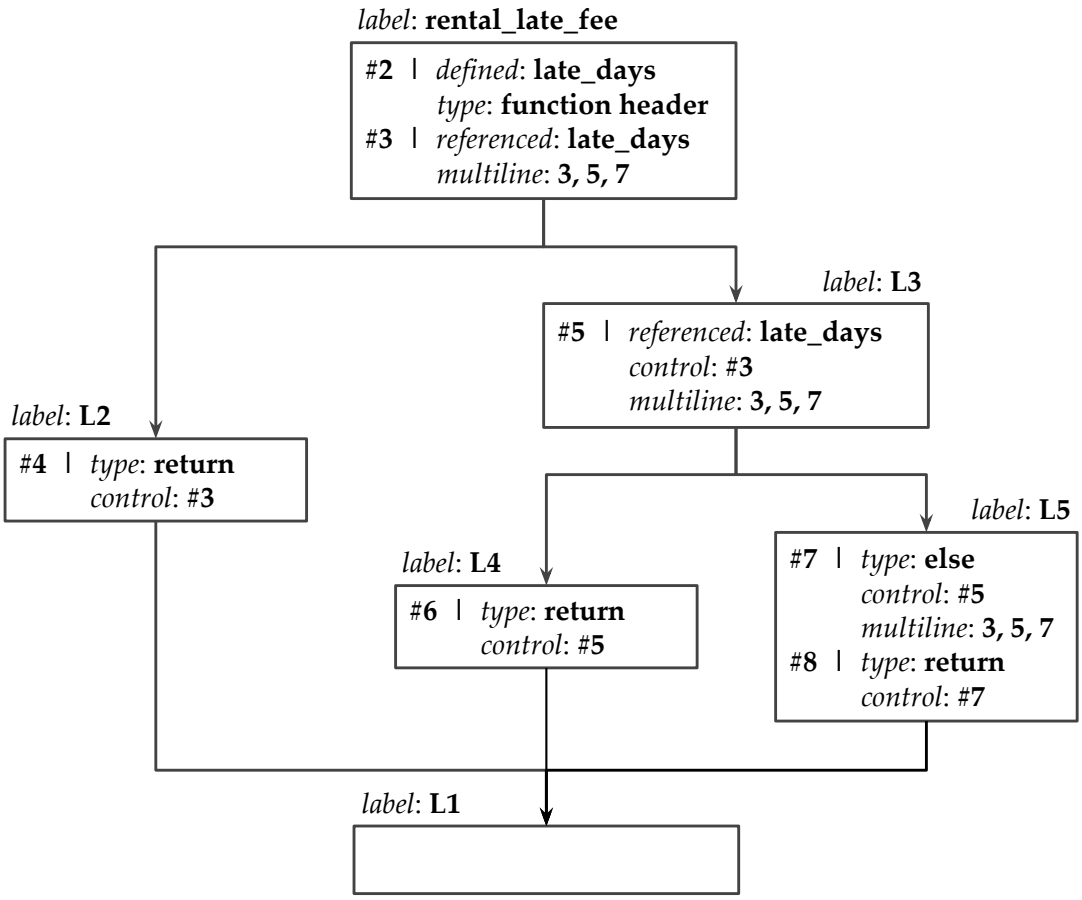

Figure 2.3: CFG containing conditionals in Listing 2.3

Listing 2.4: Nested for loop example

```
1 # Input: scores is a list of list of integers.
2 def get_total_score(scores):
3     total_score = 0
4     for row in range(len(scores)):
5         for col in range(len(scores[row])):
6             total_score += scores[row][col]
7     return total_score
```

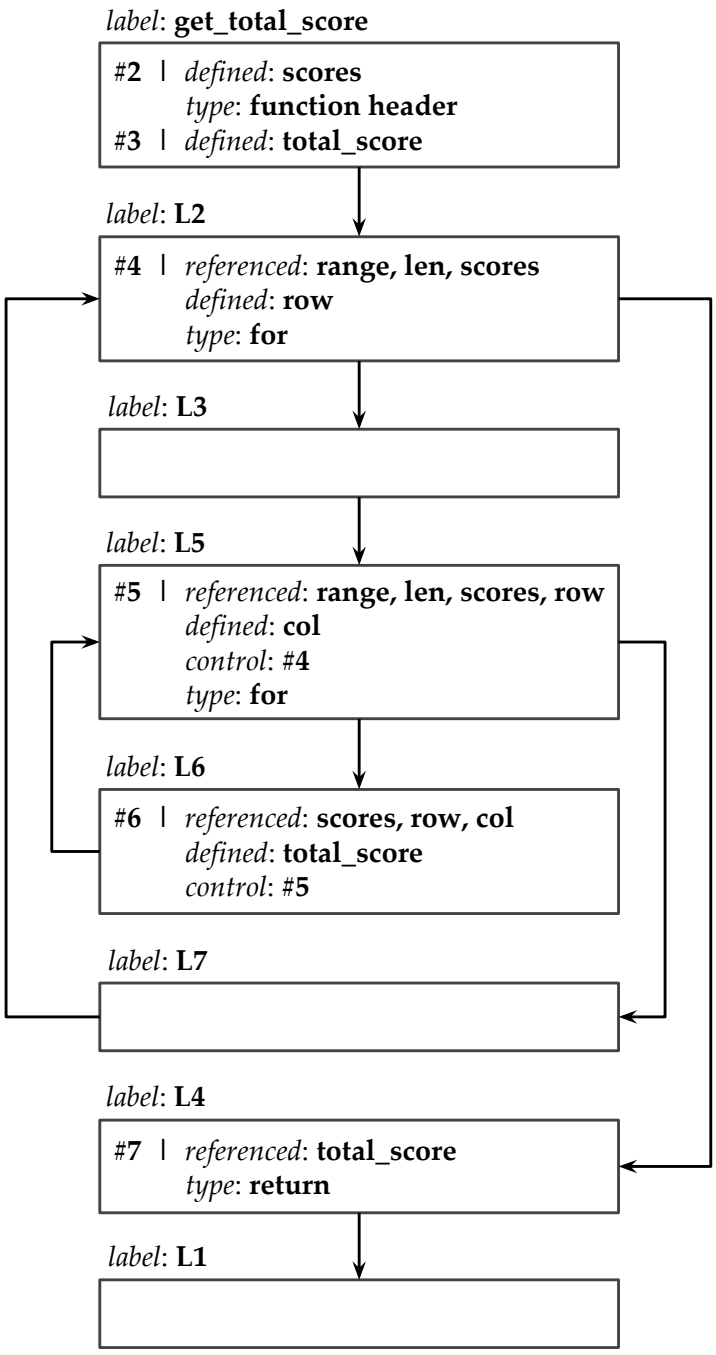

Figure 2.4: CFG containing conditionals in Listing 2.4

During CFG construction, loops have three parts: the guard block, the body block(s), and the exit block. The guard block is a stand-alone block with a single instruction - the loop condition. It also contains the edges that represent the control flow to navigate between the loop body and the code after the loop. Since there is

no code after a loop body in the case of the inner loop in Listing 2.4, an empty exit block $\texttt{L}_7$ is created. Similarly, due to a new guard block being created for the inner loop, the initial body block $\texttt{L}_3$ for the outer loop is left empty.

2.3 Iterative Data Flow Analysis

After the control flow graph is generated, iterative data flow analysis is performed to gather global information about each block and instruction to be used for further analysis. Data flow analysis provides context on how the instruction relates to the other instructions within a function. As the name indicates, the analysis is performed iteratively until the desired computation converges such that the resulting data doesn't change across an iteration [11]. *Earthworm* uses two data flow analysis algorithms: reaching definition analysis and live variable analysis.

2.3.1 Reaching Definition Analysis

Reaching definitions are the locations where a variable was defined that reach a given use of that variable [11]. For example, in Figure 2.3, the reaching definition for `late_days` referenced in both line 3 and 5 is line 2. In our case the locations can be defined by a block and an instruction number. Reaching definition analysis determines the reaching definitions for variables in each block and instruction. To compute reaching definitions going in and out of each block or instruction the program first computes the `definitions` map which helps compute the `gen` and `kill` maps for each block and instruction.

The `definitions` map maps each variable defined in the function to every location at which it was defined. The `gen` map tracks the location of the last definition of the variables within a given block or instruction. A given variable can have multiple possible definitions due to control flow such as loops and conditional statements. The

`kill` map contains all of the definitions of the variables defined in the given block. The `definitions`, `gen` and `kill` maps for the function in Listing 2.5, which contains a single basic block, can be seen in Listing 2.6 [11].

Listing 2.5: Function containing a single basic block

```
1  def funcA():
2      i = 3
3      i = k = i + 1
4      a = k + 2
```

Listing 2.6: gen and kill maps for Listing 2.5

```
1  # <variable> : set([(<block label>, <instruction number>)])
2  definitions = {'i': {('funcA', 2), ('funcA', 3)},
3                 'k': {('funcA', 3)},
4                 'a': {('funcA', 4)}}
5
6  gen = {'i': {('funcA', 3)},
7         'k': {('funcA', 3)},
8         'a': {('funcA', 4)}}
9
10 kill = {'i': {('funcA', 2), ('funcA', 3)},
11         'k': {('funcA', 3)},
12         'a': {('funcA', 4)}}
```

Listing 2.7: Single basic block function

```
1  def funcA():
2      i = 3
3      i = k = i + 1
4      a = k + 2
5      while a > 0:
6          i = i + 1
7          k = k - 1
8          if i != k:
9              a = a - 1
10         i = i + 1
```

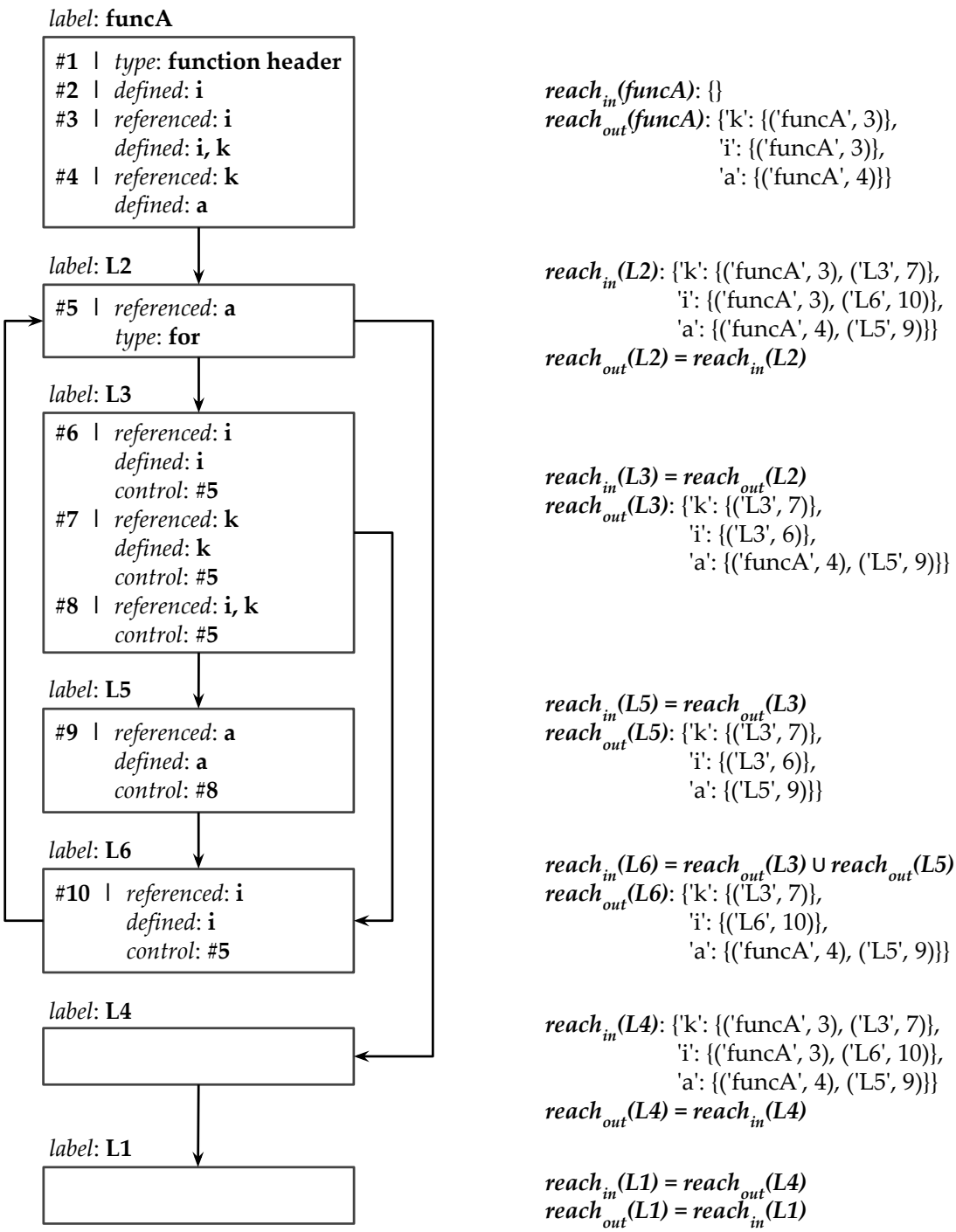

Figure 2.5: CFG with reaching definitions for Listing 2.7

The `gen` and `kill` maps can be used to compute the reaching definitions maps going in and out of the blocks and instructions using Equation 2.1 and Equation 2.2, respectively [13, 11].

$$reach_{in}(block) = \bigcup_{p \in pred(block)} reach_{out}(p) \tag{2.1}$$

$$reach_{out}(block) = gen(block) \cup (reach_{in}(block) - kill(block)) \tag{2.2}$$

Given that our graph can be cyclic, these equations are calculated until they converge. The reaching definitions resulting from the function in Listing 2.7 containing a `while` loop and an `if` statement be seen in Figure 2.5. The figure represents the reaching definitions going into a block with label *block_label* as $reach_{in}$*(block_label)* and the reaching definitions going out as $reach_{out}$*(block_label)*. The analysis is also performed on each instruction.

2.3.2 Live Variable Analysis

Live variable analysis determines the variables that are referenced after the given instruction or block. These variables are considered live because their values are still to be used, hence the name live variable analysis [11].

To compute live variables going in and out of each block or instruction, the program first computes the `variables` set, the `defined` set, and the `referenced` set. The `variables` set contains all of the variables defined in the function. The `defined` set contains all of the variables defined within a given block or instruction. The `referenced` set contains all of the variables that are referenced in a given block or instruction that have not been defined within the block before the first reference. The `referenced` set can only contain variables that occur in the `variables` set to prevent any non-local variables, such as functions, from being added to the `referenced` set. This ensures the live variable analysis, as well as other analyses using the `referenced`

set discussed in Chapter 4, is only performed on local variables.

The `defined` and `referenced` sets for Listing 2.8 are shown in Listing 2.9. Listing 2.8 contains a single basic block of a function. This code sample illustrates a few points about the live variable analysis algorithm. To begin with, `add_two` is not part of the `variables` set because it is not defined in the function. Additionally, although both `i` and `k` were referenced variables, both were defined within the block in the line immediately prior to being referenced. Therefore, neither variable appears in the final `referenced` set.

Listing 2.8: Single basic block of a function

```
1 i = n
2 i = k = i + 1
3 a = add_two(k)
```

Listing 2.9: Listing 2.8 defined and referenced sets.

```
1 variables = {'i', 'k', 'a'}
2 defined = {'i', 'k', 'a'}
3 referenced = {'n'}
```

Listing 2.10: Single basic block function

```
1 def funcA(x, y, z, c, d):
2     while c < 5:
3         x = y + 1
4         y = mult_2(z)
5         if d:
6             x = y + z
7         z = 1
8     z = x
```

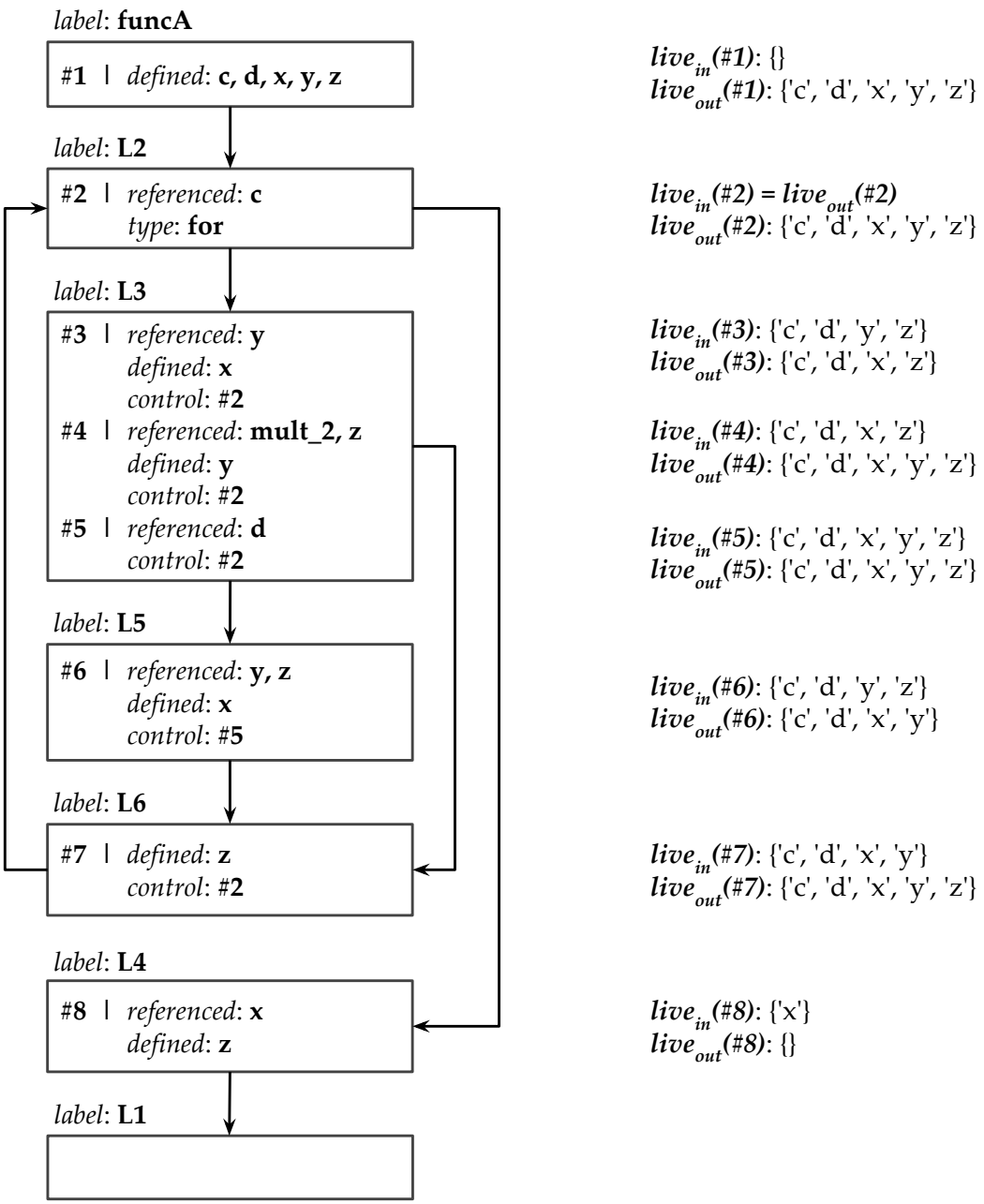

Figure 2.6: CFG with live variables for Listing 2.10

The `defined` and `referenced` sets can be used to compute the live variable sets going in and out of the blocks and instructions using Equation 2.3 and Equation 2.4, respectively [13, 11].

$$live_{in}(block) = referenced(block) \cup (live_{out}(block) - defined(block)) \tag{2.3}$$

$$live_{out}(block) = \bigcup_{s \epsilon succ(block)} live_{in}(s) \tag{2.4}$$

As with reaching definition analysis, the live variable analysis equations are calculated multiple times until they converge. The live variables resulting for each instruction in the function in Listing 2.10 containing a `while` loop and an `if` statement can be seen in Figure 2.6. The figure represents the live variables going into an instruction with line number *num* as $live_{in}$*(#num)* and the live variables going out as $live_{out}$*(#num)*. As with reaching definitions, live variable analysis is performed on the block and then the instruction level.

2.4 Program Slicing

A slice is a subset of the control flow graph that represents all instructions and blocks on which an instruction depends [19]. A slice identifies the code that can be considered for restructuring.

Listing 2.11: Example code for investigating slices.

```
1  def get_num_pixels(width, height):
2      cols = 0
3      pixels = 0
4      for y in range(width):
5          for x in range(height):
6              pixels += 1
7              new_var = 0
8          cols += 1
9      temp = 0
10     print(cols)
11     print(pixels)
```

```
1  def get_num_pixels(width, height):
2      cols = 0
3      pixels = 0
4      for y in range(width):
5          for x in range(height):
6              pixels += 1
7              new_var = 0
8          cols += 1
9      temp = 0
10     print(cols)
11     print(pixels)
```

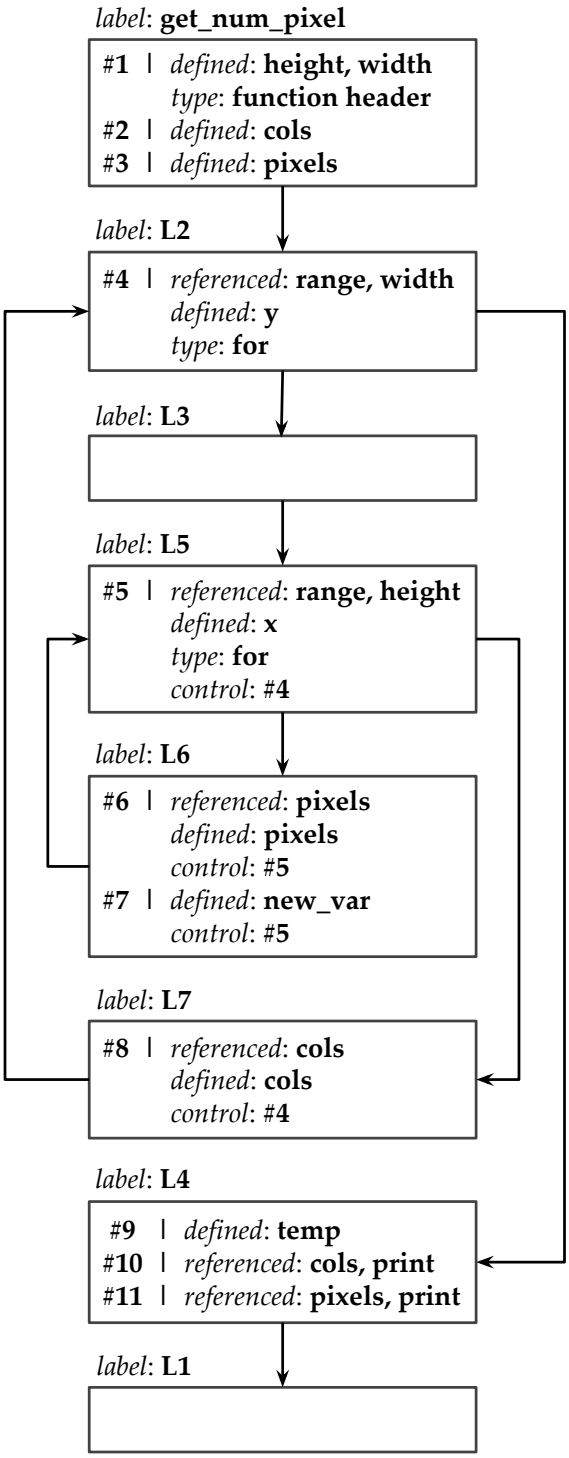

Figure 2.7: CFG for Listing 2.11

The concept of a slice can be illustrated through further examination of Listing 2.11. The function `get_num_pixels` calculates the number of pixels given the screen size of `width` by `height`. The control flow graph for Listing 2.11 can be seen in Figure 2.7.

```
1  def get_num_pixels(width, height):
2      cols = 0
3      pixels = 0
4      for y in range(width):
5          for x in range(height):
6              pixels += 1
7              new_var = 0
8          cols += 1
9      temp = 0
10     print(cols)
11     print(pixels)
```

label: **get_num_pixel**

#9	*defined*: **temp**

Figure 2.8: Condensed CFG for slice at instruction 9 in Listing 2.11

A slice of instruction number 9 consists of only one block containing `temp = 0` as seen in Figure 2.8. The assignment of `temp` to `0` indicates the value of `temp` at that instruction is not dependant on any previous lines of code. Additionally, line number 9 is not tied to a logical control point. Therefore, the final control flow graph, as seen in 2.8, contains only line number 9.

Examining these slices closely, a pattern emerges. Generating the slice on instruction number 10 in Figure 2.9 requires walking backwards through the CFG visiting each instruction where a referenced variable was previously defined. Starting at instruction number 10, `cols` was most recently defined in line 8. Prior to line 8, `cols` was defined on line 2. Instruction number 4 is added to the slice because it is the control for line 8 and affects when it will be run. Instruction number 4 contains a reference to `width` which is defined on line 1. Therefore, a slice of instruction number 10 contains instructions numbered 10, 8, 4, 2, and 1.

```
1  def get_num_pixels(width, height):
2      cols = 0
3      pixels = 0
4      for y in range(width):
5          for x in range(height):
6              pixels += 1
7              new_var = 0
8          cols += 1
9      temp = 0
10     print(cols)
11     print(pixels)
```

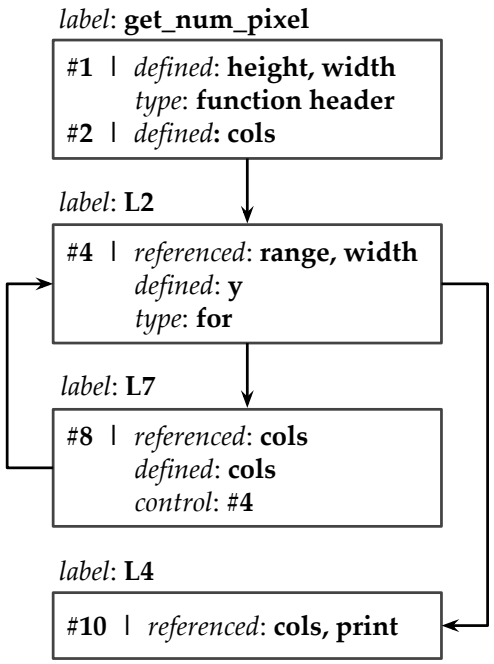

Figure 2.9: Condensed CFG for slice at instruction 10 in Listing 2.11

```
1  def get_num_pixels(width, height):
2      cols = 0
3      pixels = 0
4      for y in range(width):
5          for x in range(height):
6              pixels += 1
7              new_var = 0
8          cols += 1
9      temp = 0
10     print(cols)
11     print(pixels)
```

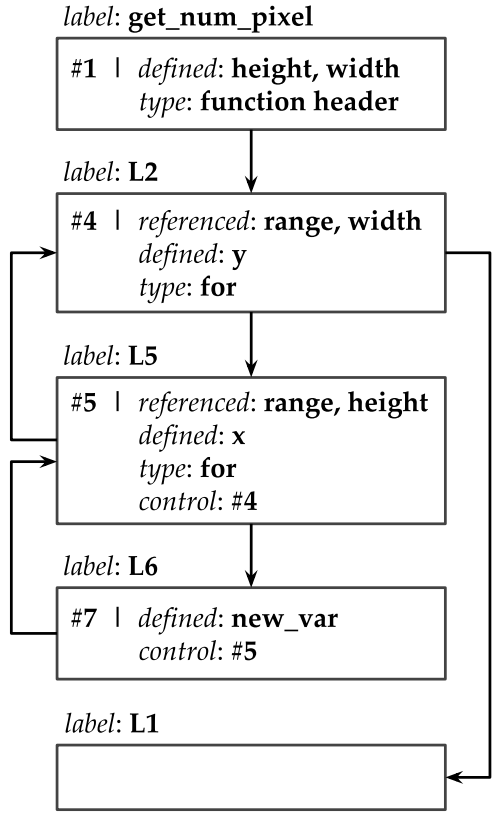

Figure 2.10: Condensed CFG for slice at instruction 7 in Listing 2.11

A slice of instruction number 7, as seen in Figure 2.10, is more complex than the slice at line 9 due to the presence of logical control in the form of nested `for` loops. Line 7 runs as dictated by line 5 which in turn is controlled by line 4. The variables referenced on lines 4 and 5 are defined on line 1. The final slice of instruction number 7 contains instructions numbered 7, 5, 4, 1.

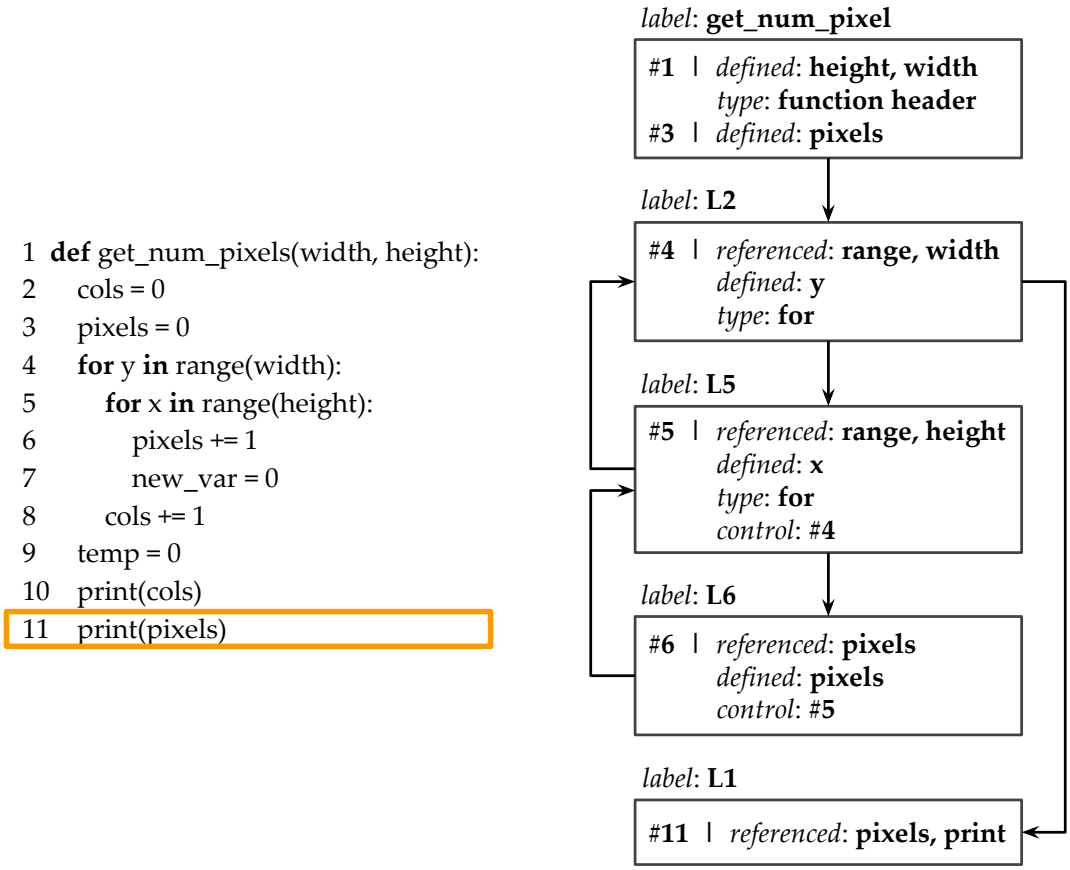

Figure 2.11: Condensed CFG for slice at instruction 11 in Listing 2.11

The largest control flow graph that can be generated from a slice of this program comes from taking a slice of instruction number 11. Due to `pixels` being defined inside the nested for loop, this slice would contain instructions numbered 11, 6, 5, 4, 3, 2, and 1. This can be seen in Figure 2.11.

A slice is generated using the definition of each variable in the reference set of an instruction and adding in the control and multiline instructions. To determine the

last definition of a variable, the program capitalizes on the previous computation of reaching definitions.

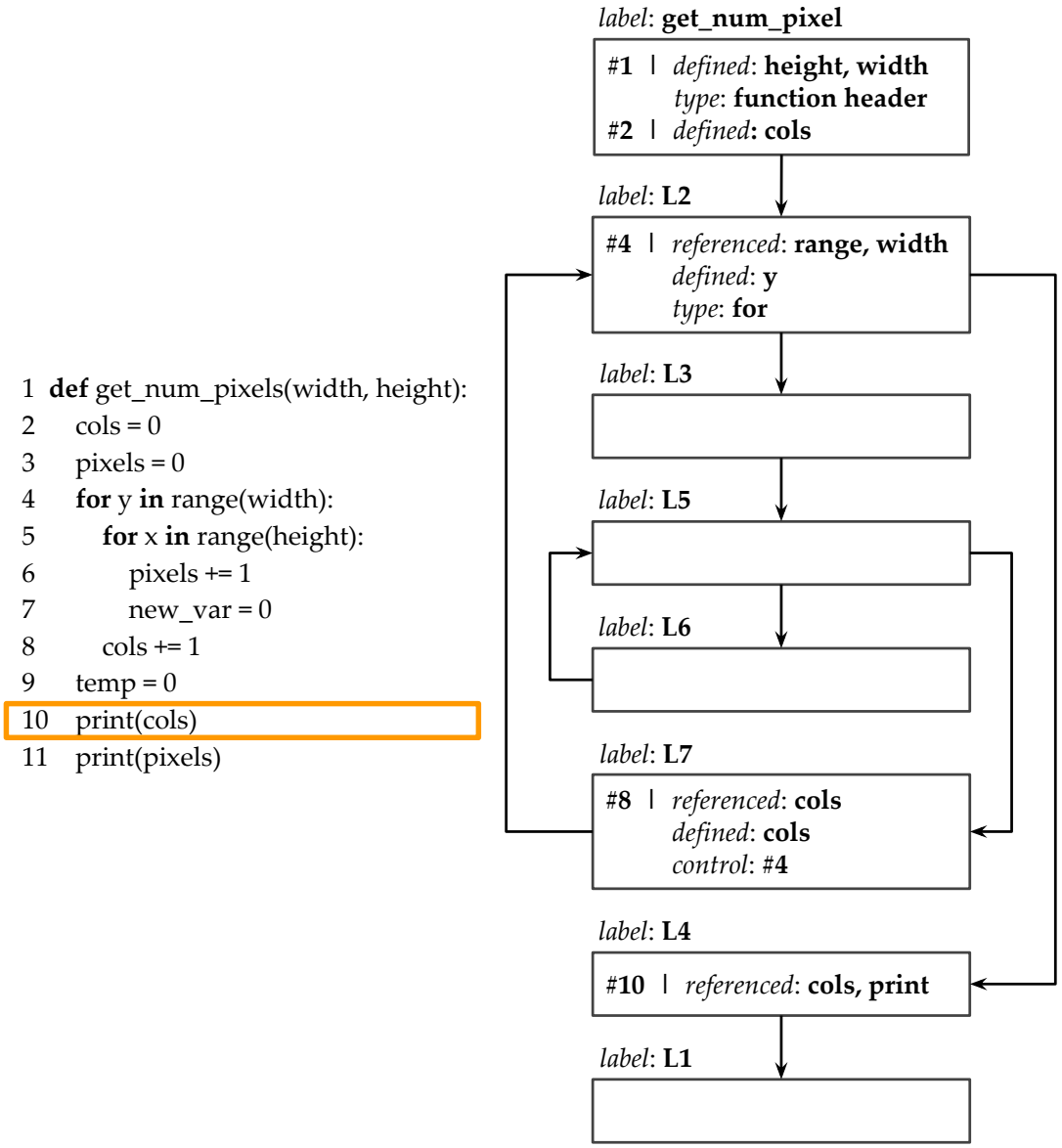

Figure 2.12: Original CFG for slice at instruction 10 in Listing 2.11

The program generates a slice at a given instruction within a control flow graph simply by accessing the stored global information for each block and instruction. However, it is important to note, the graphs depicted above are the final desired results. The slicing algorithm itself just outputs the instruction numbers of the slice to include in the control flow graph. Removing all other instructions results in a

control flow graph containing only those instructions in the slice but with additional nodes and edges. This can be seen in Figure 2.12 which represents the full control flow graph on a slice on instruction number 10 but without any additional analysis performed to remove extra nodes and edges.

2.5 Clean Algorithm

These extra paths between blocks with zero instructions can be removed by the Clean algorithm [11]. The Clean algorithm minimizes the CFG so it contains the minimum paths and blocks necessary to represent the function being depicted. Like the previous algorithms presented, the Clean algorithm is an iterative algorithm; the algorithm repeats until the CFG stops changing. The Clean algorithm supports the following transformations.

2.5.1 Remove Empty Block

When the Clean algorithm finds an empty block `B` with a single successor `S`, it makes all predecessors of `B` point to `S` [11]. In *Earthworm*, this often occurs during the construction of nested loops and when a slice removes all instructions from a block.

The example scenario of empty blocks in nested loops happens in code such as Listing 2.4. The corresponding CFG can be seen in Figure 2.4. Both the empty exit block $\texttt{L}_7$ and the empty guard block $\texttt{L}_3$ in Listing 2.4 can be removed by the Clean algorithm as seen in Figure 2.13.

Empty block removal was also performed between Figure 2.14 and Figure 2.15, removing blocks $\texttt{L}_2$ and $\texttt{L}_3$. Figure 2.14 represents the slice at line number `7` for Listing 2.12. The slice only contains line number 2, which defines `a`, and line number `7`, which prints `a`.

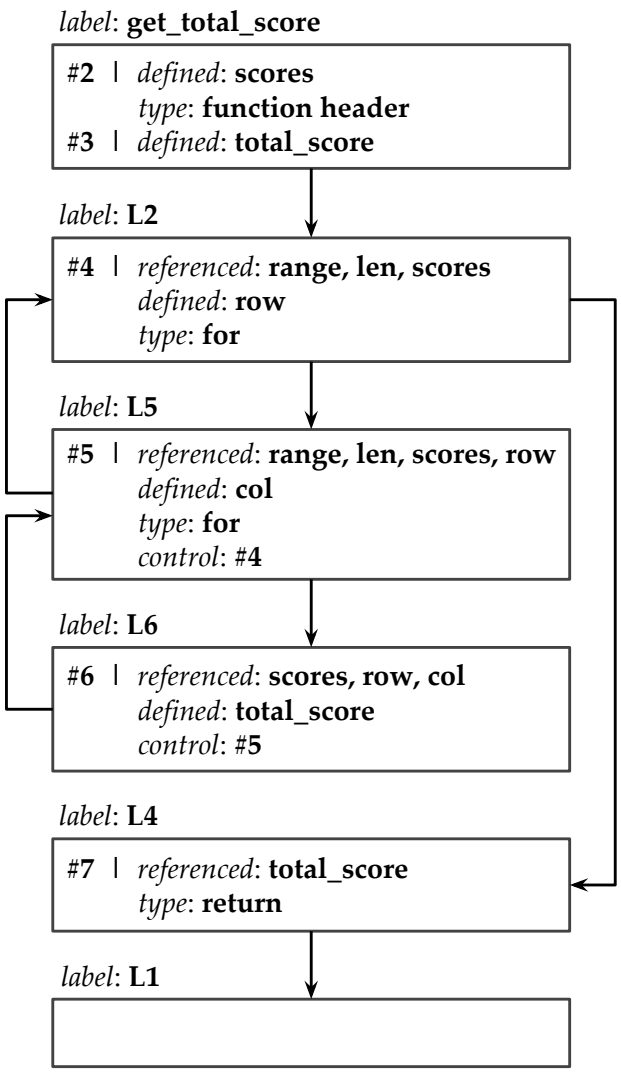

Figure 2.13: CFG of Listing 2.4 after removing empty blocks.

Listing 2.12: Example code for investing Clean algorithm.

```
1 def funcA(n):
2     a = 5
3     if n > 5:
4         print("happy")
5     else:
6         print("sad")
7     print(a)
```

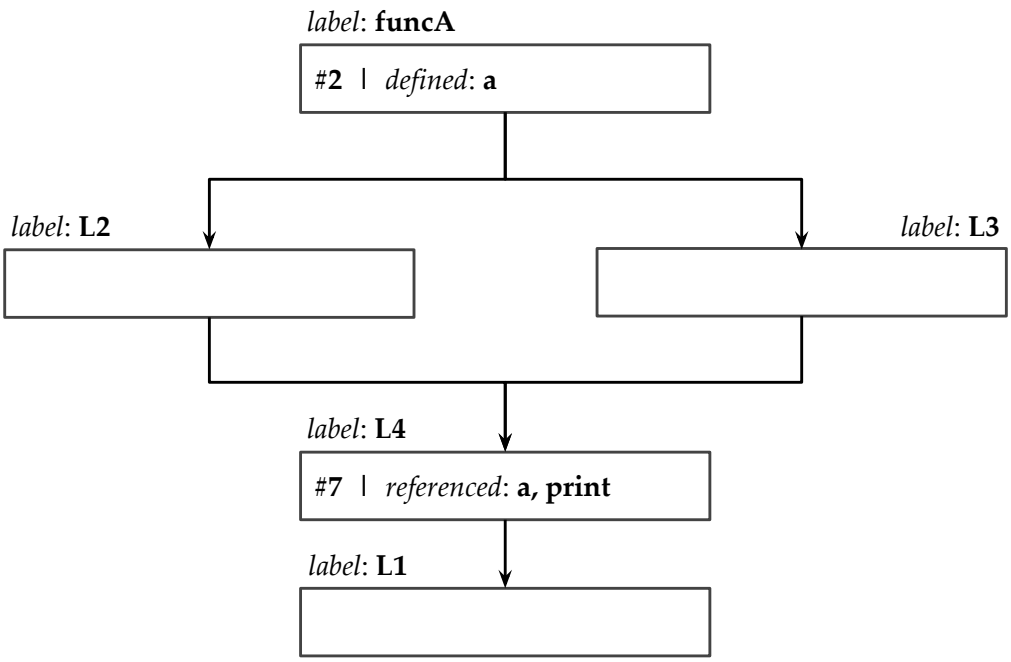

Figure 2.14: CFG for slice at instruction 7 in Listing 2.12 prior to Clean algorithm.

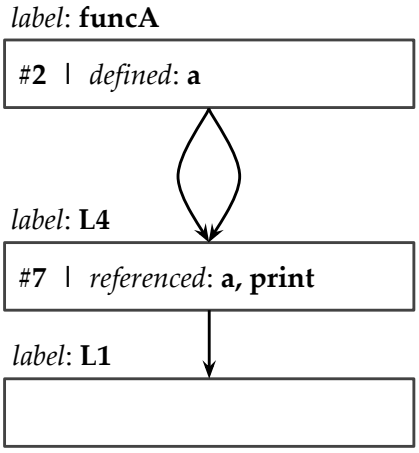

Figure 2.15: CFG following Figure 2.14 after removing empty blocks.

2.5.2 Fold Redundant Branch

When the Clean algorithm finds a block `B` with multiple references to block `S` in `B`'s successor list, it condenses them into a single reference [11]. In *Earthworm*, this often occurs after running the slicing algorithm on a CFG and removing empty blocks representing a branch.

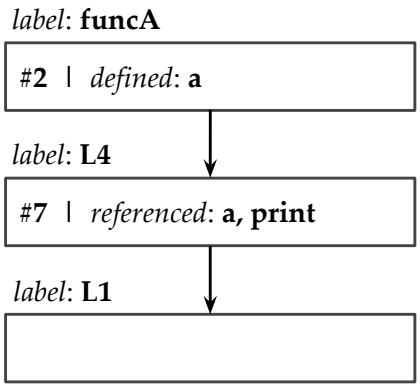

Figure 2.16: CFG following Figure 2.15 after folding redundant branches.

Redundant branch folding was performed between Figure 2.15 and Figure 2.16; the Clean algorithm removes duplicate successor branches between $\texttt{L}_1$ and $\texttt{L}_4$.

2.5.3 Combine Blocks

When the Clean algorithm finds a block `B` with a single successor, it combines the block with its successor `S`. Block `B` must be the only predecessor to block `S`. All instructions and successors of `S` are added to block `B` [11]. In *Earthworm* this primarily occurs during construction for functions with a single return statement and due to instruction removal during slicing.

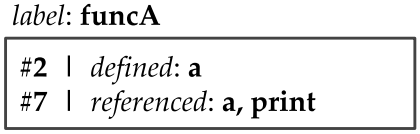

Figure 2.17: CFG following Figure 2.17 after combining blocks.

Both examples are illustrated in the transformation between Figure 2.16 and 2.17. First, block $\texttt{L}_4$, containing a single predecessor, is condensed with `funcA` containing a single successor. Next the empty sink block $\texttt{L}_1$ is merged with `funcA` generating a slice with a single block. Sink blocks such as $\texttt{L}_1$ are generated during construction in functions with one return statement. As previously mentioned, any blocks within the function that contain a return statement are set to point to the sink block. Therefore

a function with a single return statement would be constructed as a sink with a single predecessor. The sink would always be combined with its predecessor upon running the Clean algorithm.

2.5.4 Hoist Branch

When the Clean algorithm finds a block `B` with a single successor, block `S`, such that `S` is an empty block with multiple successors, the successors of block `B` are replaced with the successors of block `S`. This is best illustrated in Figure 2.18 adapted from *Engineering A Compiler* [11].

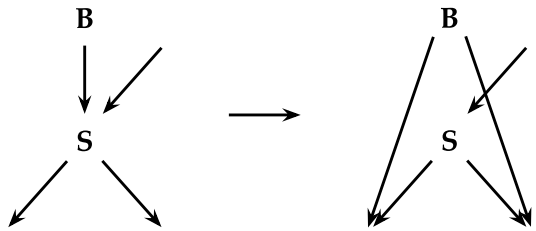

Figure 2.18: Figure illustrating hoisting a branch.

Listing 2.13: Example code for investing hoisting a branch.

```
1 def funcA():
2     a = 5
3     for num in range(5):
4         print(num)
5     print(a)
```

Branch hoisting did not occur as commonly in my code due to my usage of slicing. It is primarily used to remove a loop assuming code prior to it and after it is included in the slice while the loop itself is excluded. Branch hoisting is performed between Figure 2.19 and Figure 2.20. Figure 2.19 represents the slice of Listing 2.13 at line number 2, which defines `a` and line number 5 which prints `a`. Since `funcA` has a single successor L_2 that is an empty block with a branch, the Clean algorithm replaces the

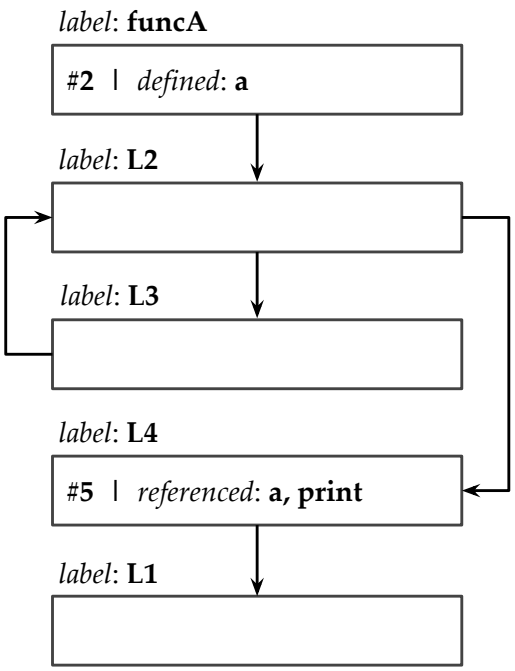

Figure 2.19: CFG for slice at instruction 7 in Listing 2.13 prior to Clean algorithm.

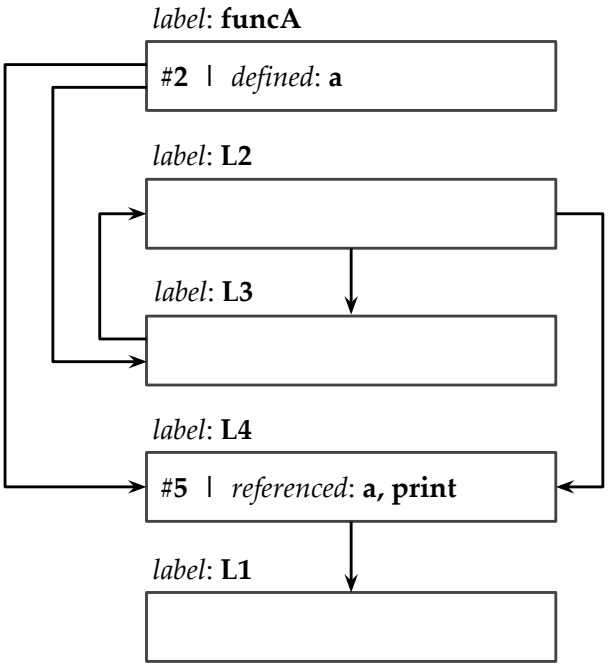

Figure 2.20: CFG of Listing 2.13 after hoisting a branch.

successors of `funcA` with the successors of $\texttt{L}_2$. The other three parts of the Clean algorithm will then reduce the slice into a single block similar to Figure 2.17.

2.6 Cyclomatic Complexity Algorithm

One commonly used quantitative measure of complexity is cyclomatic complexity. Cyclomatic complexity measures the number of linearly-independent paths through a program represented as a CFG [17]. A linearly-independent path, in the context of CFGs, is a path through the program that introduces a block that is not in any other linearly-independent path.

Cyclomatic complexity `M` can be computed using Equation 2.5 where `E` represents the number of edges, `N` represents the number of nodes, and `P` represents the number of nodes with exit points [17]. Exit points are blocks with more than one predecessor or the sink block.

$$M = E - N + 2 * P \tag{2.5}$$

Since cyclomatic complexity is based on graph structure, the Clean algorithm helps with more accurately evaluating changes in complexity due to program slicing.

2.7 Summary

This chapter discussed various algorithms and data structures that will be used to generate suggestions on source code. The chapter first introduced control flow graphs, which represent student's source code as a directed graph with a source and sink node. It then discussed two types of iterative data flow analyses: reaching definition analysis and live variable analysis. Reaching definition analysis gets the locations where a variable is defined that reach a given use of each variable. Live variable analysis determines the variables that are referenced after the given instruction or block. Next, the chapter introduced program slicing which finds a subset of the control flow graph that represents all instructions and blocks on which an instruction depends. The Clean algorithm minimizes the CFG so it contains the minimum paths and blocks

necessary to represent the function being depicted. Lastly, the chapter introduced cyclomatic complexity, which measures the number of linearly-independent paths through a program represented as a CFG.

Chapter 3

RELATED WORK

This chapter discusses refactoring and linting tools, what they are, and their relation to *Earthworm*. Additional applications of program slicing are also discussed.

3.1 Refactoring Tools

Refactoring is the process of restructuring code, a concept previously introduced in Section 2.1. Refactoring tools automate aspects of refactoring - easing tasks that can be tedious and time consuming. This section examines three different IDEs that each provide a set of refactoring tools for a particular language. Most refactoring tools, including the ones listed below, primarily focus on simplifying the task of refactoring after the developer has identified what they want to refactor. None of the IDEs contain tools that make suggestions on how best to decompose the code.

3.1.1 PyCharm

PyCharm has a suite of refactoring tools for Python. Some of the features related to *Earthworm* are listed below [4].

- *Change Signature* - Changes the name of a function or the parameters being passed into a function.
- *Extract* - Extracts a section of code within a function to create another function. Extracts part of an instruction into a constant, field, variable, or parameter.
- *Inline* - Folds a variable into the uses of the variable.

- *Rename* - Renames any symbols including variables, functions, fields, and parameters.

As previously noted, many of these features help ease the process of refactoring code. However, they require the developer to be knowledgeable about the best means of refactoring the code. The features provided by PyCharm, particularly *Extract* and *Rename*, can be used in conjunction with *Earthworm* to ease the process of refactoring.

3.1.2 Eclipse

Eclipse contains a similar suite of refactoring tools for Java including *Rename* and *Extract*. In addition to the tools provided by PyCharm, Eclipse contains a few additional tools for generating code following specific design patterns that are commonly used in Java [3].

3.1.3 Visual Studio

Visual Studio contains refactoring tools for C++, C#, Visual Basic, and F# that parallel those in both PyCharm and Eclipse [9].

3.2 Python Linting Tools

A linter typically flags code that does not follow the construct of the language as identified in the formal documentation. In Python this formal documentation is `PEP 8`. In a broader sense, linting tools help identify stylistic issues in a program. Student code in introductory classes tends to contain poor stylistic choices. Therefore, building `Earthworm` to have a basic linter would enable students to write more readable code. Two exisiting Python linters are Pylint and PyFlakes.

3.2.1 Pylint

Pylint is the most widely known style checker for Python. Pylint strictly adheres to the Python `PEP 8` style guidelines. It provides an extremely comprehensive output of instructions in the code that fail to conform to the style guidelines. In addition, it provides a rating with a maximum score of 10.0 and no minimum score to help summarize how the code fares in comparison to the `PEP 8` guidelines [6].

Unfortunately, this presents the disadvantage of it being extremely strict and potentially intimidating to a beginning coder. The suggestions can be confusing at times and very verbose. For example, Pylint outputs every instruction where the indentation at that indentation level is not 4 spaces from the previous indentation level.

Overall, many of Pylint's suggestions can be useful. However, for beginning coders it can be intimidating and overly strict.

3.2.2 Pyflakes

Pyflakes was intended to more loosely enforce Python style guidelines. However, Pylint only provides basic name errors where class names are not found and identifies unused imports or `import *` as poor Python practice [5]. For a classroom setting it would be very limiting on what it suggests as it finds most programs to be perfect.

3.3 Other Approaches to Program Slicing

Program slicing was originally introduced by Mark Weiser in his paper, *Program Slicing* [19]. Weiser formalized the process his students were following as they debugged, defining a process and its outputs. He introduced slicing as a potential use for debugging, testing, and parallel processing. In his paper he discussed his imple-

mentation of program slicers to find useless instructions within Fortran programs. Since then slices have been used in various applications.

The application most similar to *Earthworm* is that of Lakhotia and Deprez [14] which used slicing to develop a system, *Wolfpack*, that restructured C functions with low cohesion into functions with high cohesion. Cohesion is the degree to which elements in a module belong together. Slicing was used to identify statements necessary for output variables with high cohesion in a function.

Other applications of slicing[15, 16, 18] primarily used to develop tools for developers to fix existing code. *Earthworm* examines the subject with a different angle of using slicing to generate suggestions with the purpose of educating. Additionally, *Earthworm* is aimed at introductory Python code while the tools mentioned above were intended for larger legacy code in languages that are largely obsolete in computer science education (particularly introductory education).

Chapter 4

IMPLEMENTATION

This chapter presents the design of *Earthworm* and the approaches used to generate suggestions on code samples. First, the requirements for *Earthworm* are stated.

4.1 Requirements

Earthworm was developed with the intention of assisting students enrolled in introductory computer science courses at Cal Poly. In order to cater to our introductory series, which begins in Python, the requirements for *Earthworm* are.

1. Accepts Python 2 and 3 code.
2. Easy to run on a single file.
3. Generates suggestions within a reasonable amount of time (less than 60 seconds) for a small program (less than 200 lines of code).
4. Handles basic Python instructions for each function or method:
    - variable assignments and references
    - `return` statements
    - conditionals
    - `for` and `while` loops
    - exceptions
    - `pass`, `continue`, and `break`
    - basic list functions: `append`, `insert`, `extend`, and `pop`

5. Individualized feedback on line numbers to decompose.

6. (optional) Give feedback on stylistic issues (e.g. indentation changing between indentation levels in a function)

4.1.1 Common Issues

Prior to developing the program, I investigated common issues in programs developed by introductory programmers. I examined a suite of student programs developed by students in previous CPE 101, Introduction to Computer Science, courses taught at Cal Poly. The code commonly contained the following issues:

- Functional decomposition was limited to what was provided by the specification although more was warranted.
- Significant portions of the program were written within control flow such as a conditional or as a loop body.
- Multiple lines of code referenced the same variables in a similar fashion and were contained within a larger function.
- Loops performing a single task and returning a single variable were contained within a larger function.
- Inconsistent tabbing across functions or across indentation levels within a function.
- Repetition of code across functions.
- Very long functions.

4.1.2 Approach

Taking these requirements into consideration, I developed a program that takes in a single Python file with one or more functions or methods and generates zero or more suggestions on each function. Taking the common issues into consideration, the suggestions are generated using the following three approaches described in detail in the subsequent sections:

1. Find continuous sets of instructions where removing variables from the slice decreases a slice's cyclomatic complexity.
2. Determine multiple consecutive instructions using similar references.
3. Examine differences in live variables and references within a block or instruction.

Although these approaches do not tackle all of the common issues seen in student code, they take on a significant portion of them as outlined in the following three subsections. Additional approaches considered for future work are discussed in Section 7. In the following discussion, many of the parameters discussed can be adjusted via a configuration file.

4.2 Reduced Slice Complexity from Removing Variables

The first approach *Earthworm* uses is to examine differences in slice complexity before and after removing variables from slices. The purpose of the approach is to identify instructions containing variables controlled by a significant amount of control flow.

Earthworm contains two functions: generate a slice map and generate variable groups. The **slice map** maps each instruction in the function to a slice at that

instruction. The slices are run through the clean algorithm to ensure they only contain the necessary blocks and edges. **Variable groups** are groups of one or more variables that are defined at some point in the function.

The initial implementation of *Earthworm* generated all variable groups of size 1 through 5. However, this slowed down the program measurably, beyond what was permissible by our requirements. The current implementation generates variable groups based on variables defined and referenced near each other. The idea is motivated by the assumption that two variables accessed far apart are less likely to be part of control flow that decreases slice complexity at a given instruction. Additionally, only variable groups of size 1, 3, and 4 remain. A qualitative analysis of the suggestions demonstrated groups of 2 and 5 added additional time and noise instead of producing high-quality, unique suggestions not already covered by the other variable group sizes. A sampling of the variable groups for Listing 4.1 are given in Listing 4.2.

Listing 4.1: Example code for investigating suggestions based on reduced slice complexity.

```
1 def get_num_pixels(width, height):
2     pixels = 0
3     for row in range(height):
4         for col in range(width):
5             pixels += 1
6     print(pixels)
```

Listing 4.2: Sample variable groups for Listing 4.1

```
1 set(width, height, pixels, row, col,
2     (width, height), (height, pixels), (pixels, row), ...
3     (width, height, pixels), (height, pixels, row), ...)
```

Once the variable groups are generated, the program generates a slice map for each variable group. In normal slices, the slice is generated by finding the last definition of each variable referenced in any instruction being added to the slice. Here, however, when a variable group is provided, any referenced variable in the variable group is

ignored - the reference is not traced back to its definition.[1]

The program compares the slice map of the original code with the slice map generated from each variable group. The program keeps a list of any instructions where the slice for a variable group decreased in complexity when compared with the original slice map.[2] The code complexity is determined using cyclomatic complexity.

This slice comparison process can be seen more clearly when examining the slice of Listing 4.1 at instruction 6. The original slice, seen in Figure 4.1 has a cyclomatic complexity of 7. Meanwhile a slice map with variable group of `'pixels'` consists of a single block containing instruction number 6 as seen in Figure 4.2. The only reference in instruction number 6, `pixels`, is part of the referenced variables. Therefore its reaching definition is not added to the slice. The revised slice has a cyclomatic complexity of 1. Since the difference in slice complexity between the slice for the variable group and the slice for the original map is greater than 3, this line number would be added to the current list of instructions for the given variable group.

Once all of the instructions with a lower complexity for a given variable group are identified, the instructions are then grouped into suggestions based on the criteria outlined in section 4.5.

It is of value to note that any instructions that are the control for any instructions in the slice, are added even if the referenced variables in that instruction are part of the variable group. This can be seen in Figure 4.3 which shows a slice at instruction 5 with variable group `'pixels', 'width', 'height'`. At instruction 5, the only referenced variable, `pixels`, is part of the variable group. Therefore, instruction 1 is not added to the slice although it is one of the reaching definitions of `pixels`.

[1] Throughout the process, the CFG generated from the list of instructions identified as constructing a slice are cached. This saves time in lieu of space. This is permissible due to the small size of the programs being run and the desire for higher time performance.

[2] In the examples provided in this paper, the difference in complexity had to be greater than or equal to 3.

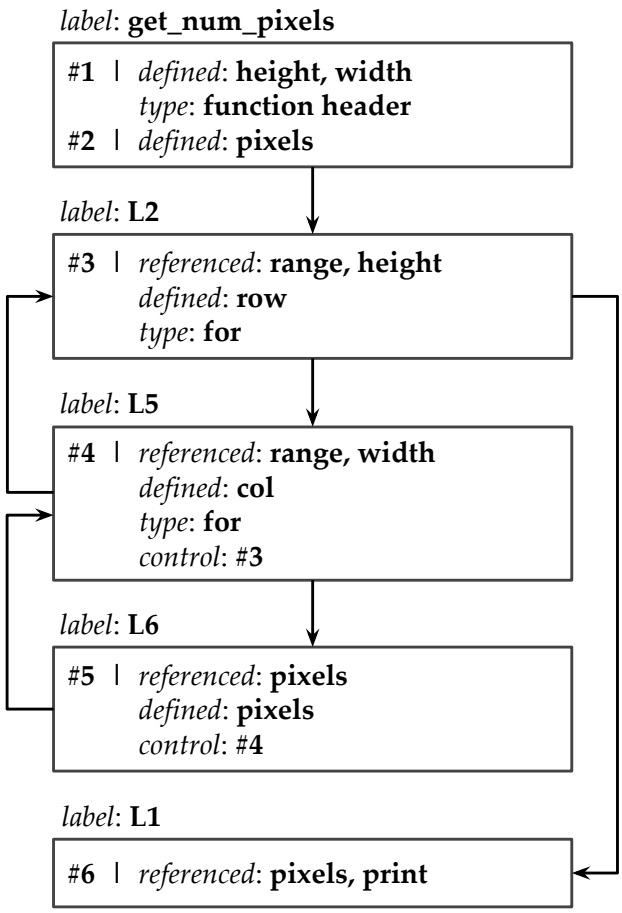

Figure 4.1: CFG for slice at instruction 6 for Listing 4.1

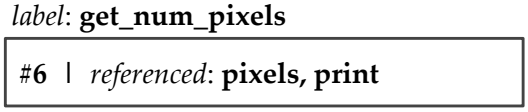

Figure 4.2: CFG for slice at instruction 6 with variable group 'pixels' for Listing 4.1

However, the blocks `L2` and `L5` are still included because instruction 3 and 4 control instruction 4 and 5 respectively, resulting in a cyclomatic complexity of 7. Therefore, this approach is not as useful in identifying instructions that are contained within control flow, but instead identifies instructions that are contained after and affected by prior control flow.

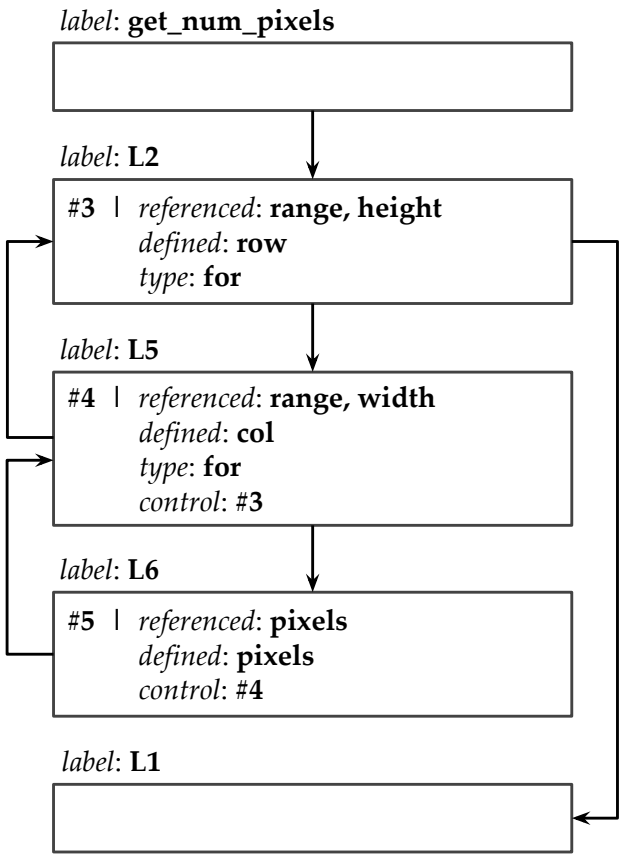

Figure 4.3: CFG for slice at instruction 5 with variable group 'pixels', 'width', 'height' for Listing 4.1

4.3 Similar References Consecutive Instructions

The second approach *Earthworm* uses to generate suggestions is to examine similarities in the variables referenced across consecutive instructions. The goal of this approach is to identify multiple successive instructions referencing the same set of variables in a generally similar fashion.

The program iterates through the instructions and groups those where consecutive instructions reference the same variables. To account for function names being included in an instruction's reference set, the reference set used is that determined by live variable analysis.

After the groups of suggestions are identified, the groups are based on the criteria

outlined in 4.5.2 to ensure only viable suggestions remain.

4.4 Differences in Live Variables and Referenced Variables

The final approach *Earthworm* uses to generate suggestions is to examine the differences in the live variable set and reference set for blocks and instructions. The purpose of the approach is to identify instructions with many unused live variables. Unused live variables are variables that are live at an instruction or block but not referenced by it. Many unused live variables indicate those instructions are loosely related to the surrounding instructions.

Blocks where the live variable sets going into the block contained four or more variables than the referenced sets are identified. All instructions from identified blocks were grouped into suggestions based on the criteria outlined in Section 4.5.

Similarly, instructions where the live variable set going into the instruction contained four or more variables than the referenced set are grouped together. Due to significant noise created during the instruction-level analysis, any suggestions generated by the instruction-level analysis containing fewer than five instructions are removed.

4.5 Criteria for Grouping Line Numbers into Suggestions

The program takes two steps to convert a list of line numbers into a list of suggestions. First the line numbers are split into initial groups primarily based on adjacent line numbers. Then the preliminary groups are further split into groups that can be removed without changing the meaning of the code.

4.5.1 Criteria for Grouping Line Numbers into Groups

Prior to grouping line numbers, additional instructions that are not important to code evaluation are added to the list of line numbers. For all multiline statements represented in the list of line numbers, any line numbers in the multiline group not in the control flow graph are added. This situation primarily arises with multiline groups containing empty lines, comments, strings or lines of code with only operators and static constants. All such lines are referenced in the multiline groups but not contained in the CFG.

Next, any unimportant instructions are added to the list of line numbers. Unimportant instructions are any blank lines or comments contained in the source code. Adding unimportant instructions ensures the final suggestions aren't affected by the source codes' spacing or comments.

The final set of line numbers is then split into groups based on consecutive lines where the number of instructions is greater than one. For example, the list of instructions `[2, 3, 5, 6, 7, 9]` would be grouped into a list of tuples, `[(2, 3), (5, 7)]`, that represents the minimum line number within the group and the maximum line number within the group.

After the groups have been generated, the groups are processed as described in the following section.

4.5.2 Criteria for Generating Suggestions from Groups of Line Numbers

The groups are split up to ensure each suggestion could be easily implemented by a programmer without altering the meaning of the code. Prior to adding this post processing step, many of the outputted suggestions had to be slightly adjusted prior to implementing them in code. For example, previously the instruction containing the

`if` clause of a conditional would be included in a suggestion where the full conditional body was not included.

The program contains four criteria for splitting up suggestions. First, suggestions are split up if the indentation level tabs out out one level. This could occur if part of a conditional body was included as well as the code after it - without including the conditional clause.

Groups are also adjusted to ensure all mutiline statements are contained within a given group. For more challenging projects, some introductory students create complex multiline instructions spanning up to five lines. This originally led to suggestions that only contained a part of the instruction. This criteria removes any instructions that are not included with their full multiline group. This criteria is also successful at keeping conditionals (which are considered multiline statements for the purposes of this analysis) as one unit. Previously, the `if` clause and body would be included in the suggestion while the `else` would not be (or vise versa). Given how tightly coupled a conditional body is, they should not be split up.

Third, if all the items within the control group aren't included, then the control itself cannot be included. This handles the case mentioned at the start of this section, where suggestions often included the `if` clause and part of the body, but not the whole body. However, separating an `if` clause from part of the body is not a feasible suggestion.

Lastly, any unimportant instructions at the top or bottom of suggestion groups are removed to keep the suggestions concise. Additionally, this prevents any suggestions containing only empty lines or comments from forming.

These conditions are repeatedly checked until the suggestions converge and there are no changes to the suggestions being generated.[3]

[3]The starting groups are cached along with the final suggestions to improve performance.

4.6 Criteria for Eliminating Suggestions

After the suggestions have been formed, the suggestions are checked to see if they meet specific criteria that designate them as good suggestions. The suggestion is either kept or removed - no further consideration is given to whether the suggestion could be split up in such a way to make it a good suggestion.

Although the program might be able to generate additional high-quality suggestions, based on qualitative analysis of the suggestions (by considering every valid combination of the suggestions) doing so would increase the time complexity by a significant factor which is less desirable.

Suggestions are identified to be good suggestions if they meet all the following criteria at the time of decomposition:

- Take at minimum one variable as a parameter to remove suggestions where all the instructions are variables being set to constants.
- Have a maximum of three return variables.
- Have a minimum of three important instructions; meaning at minimum three instructions that are not newlines or comments.
- To prevent suggestions from being generated for smaller functions or functions that don't need decomposing, the input parameters cannot be identical to the function's parameters and the number of lines within the function that are not within the suggestion have to be greater than five[4].

As previously indicated, these parameters can easily be adjusted in the configuration file. The values listed above are part of the default configuration and are the

[4]This number was determined through a qualitative analysis of the suggestions with a minimum required number of lines of 3, 5, 7, and 10; it is configurable via the configuration file.

values used when the suggestions discussed in Chapter 5 were generated.

4.7 Additional Approaches Examined

Aside from the approaches listed above, I examined a few other approaches to generating quality suggestions.

With cyclomatic complexity there was no real correlation between good decomposition and the resulting complexity (high cyclomatic complexity might correlate with poorly decomposed code, but cyclomatic complexity is not technically a decomposition metric). Long pieces of straight line code would have lower complexity than shorter pieces of code with control flow. I tried to combat these issues by calculating the complexity of a CFG by getting the sum of multiplying the cyclomatic complexity of the slice at each instruction by the instruction number relative to the start of the function. While the number increased with longer functions - combating the issues with cyclomatic complexity, the result was highly relative to the amount of control flow in the function and in the program overall. Therefore, it could not be used for any top level score or metric. It could also not be used for any of the analysis methods above due to its sensitivity to changes in function length that often occur when calculating a slice on instructions.

As another approach to generating suggestions, I tried to find a continuous set of instructions where removing control flow decreases slice complexity. The idea came in conjunction with the idea of generating slices by removing variable groups. The goal was to find instructions contained within control flow that was loosely related to the instruction itself. Based on observation, some of the control flow exists just to determine when or how many times to perform the action. However, this approach was only able to identify `if` or `else` statements where all variables were set to a constant.

Lastly, I considered a few different approaches for finding complex subgraphs that did not reach the development stage. I researched the Ford-Fulkerson network flow algorithm. However, it was deemed difficult to determine useful weights and to isolate something with two inputs which was often the case in my subgraphs. I also considered determining the complexity of control flow such as loops, conditionals or exception handling statements to see if those should be pulled out into suggestions. However, I wanted *Earthworm* to be able to identify suggestions on a more general scale, instead of focusing in on specific hard-coded instruction types. I believe decomposition is more than just identifying loops and conditionals and putting them in another function, so I wanted *Earthworm* to communicate the same message through its suggestions.

4.8 Linter

Python linters, as mentioned in Section 3.2, provide information on how well the code adheres to style guidelines. In addition to generating suggestions, *Earthworm* is a linter. Many of the issues reported by *Earthworm's* linter are common issues seen in CPE 101 source code or are features that are not cleanly handled by *Earthworm*.

First, the linter identifies general language-agnostic stylistic conventions that are not met. Line lengths over 80 characters are reported. Additionally, poorly named variables are reported. To keep the linter simple, poor variable names are considered to be any variable names which contain a single type of character (e.g. `x` and `XXX`).

The linter also helps identify unsupported functionality. `try`, `for`, and `while` statements with `else` clauses are reported as poor practice for beginner code and, consequently, are not handled by *Earthworm*. In such cases, *Earthworm* continues to run with a warning. Functions within functions report an error and exit the program due to breaking functionality further in the program.

The linter identifies discouraged coding practices that are commonly seen in CPE

101. Conditionals containing an `if` and `else` branch that return `True` and `False` are idenified as seen in Listing 4.3. Additionally, any `if` conditionals containing a boolean return statement followed by the boolean return statement are also identified as seen in Listing 4.4.

Listing 4.3: Example code for if / else with boolean return

```
5 if is_found:
6     return True
7 else:
8     return False
```

Listing 4.4: Example code for if with boolean return.

```
5 if is_found:
6     return True
7 return False
```

Lastly, the Linter identifies poor indentation practices. The program reports an error and exits if there is a change in indentation for a given indentation level within a program. The program also identifies changes in indentation levels between functions as well as programs without three to four spaces as a tab. Any issues with function level indentation are reported once per function. Any issues with program level indentation are reported once for the program.

Chapter 5

ANALYSIS

This chapter presents the suggestions generated by *Earthworm* using the methods outlined in Chapter 4. It examines specific suggestions that each highlight a particular part of the implementation and explains how and why they were generated. The source code used in this section is a combination of the files used in the surveys discussed in Chapter 6 and segments of CPE 101 project submissions that represent an average student's code. Each suggestion presented was selected because it represents a particular strength or weakness of `Earthworm`.

5.1 Reduced Slice Complexity from Removing Variables

Section 4.2 presented a technique to reduce slice complexity via variable extraction. In this section, the quality of suggestions generated using this technique is analyzed. Recall that a slice map is a mapping from line numbers to slices and a variable group is a group of one or more variables that are defined in close proximity within a function. This approach compares the slice map of the original code with the slice map generated from each variable group.

The code used to analyze this approach, provided in Listing 5.1, is a subset of a program that blurs an image formatted as a PPM[1]. The code, modeled after code developed by a CPE 101 student, is part of a program that contains one function, `main`, consisting of 100 lines of code. The full program this code was taken from was also used for analysis in the surveys mentioned in Chapter 6. There are additional suggestions generated on the full program that are not discussed in this section to

[1]The full program is available in Appendix A.1.1

focus this section on providing a detailed examination of select examples.

Listing 5.1: Example code for investigating reduced slice complexity suggestion generation.

```
8       # Open file.
9       try:
10          if len(argv) == 2:
                ...
15          elif len(argv) == 3:
                ...
20          else:
                ...
23      except IOError:
            ...

27      # Print header.
28      header = inFile.readline()
29      w_and_h = inFile.readline().split()
30      width = int(w_and_h[0])
31      height = int(w_and_h[1])
32      MAX_COMP_NUM = int(inFile.readline())
33      outFile.write(header + str(width) + " " + str(height) + ...)
34
35      # Read file.
36      pixels = []
37      rgb = []
38      for line in inFile:
39          line = line.split()
40          for comp in line:
41              if len(rgb) != 3:
42                  rgb.append(comp)
43              if len(rgb) == 3:
44                  pixels.append(rgb)
45                  rgb = []
46
47      picture = []
48      i = 0
49      for row in range(height):
50          row = []
51          for col in range(width):
52              row.append(pixels[i])
53              i += 1
54          picture.append(row)
```

Four suggestions were generated on this subsection of the `main` function. At the start of this subsection of code, the conditional nested within the exception handling

block defines the value of `inFile`. Therefore, much of the code located after the exception handling that references `inFile` decreases in complexity if `inFile` becomes a parameter. There were no suggestions generated on the exception handling portion of the code for reasons detailed in Section 5.4.

The suggestions provided in Listing 5.2, Listing 5.3, and Listing 5.4 were each generated based on the variable group of `inFile`. In the 'reason' given for each of these suggestions, the paths of execution, more commonly referred to as control flow paths, are all of the paths that might be traversed through a program during execution.

Listing 5.2: Suggestion resulting from Listing 5.1.

```
1 line 28-33 (main):
2         parameters: inFile, outFile
3         returns: height, width
4         reason: Removing these instructions decreases number
5                 of paths of execution in the function -
6                 making the code more readable and testable.
```

Examining the first suggestion from Listing 5.2, making `inFile` a parameter reduces the complexity at those instructions. For example, in the current code, a slice on line 28 requires including all of the control flow from line 8 to line 26 that generates `inFile` as part of the slice. However, if `inFile` is in our variable group, the reaching definition for `inFile` is no longer added to the slice. Therefore, all of the control flow from lines 8 to 26 are not included in the slice - decreasing the slice complexity from 4 to 1. Instead of having four possible paths the code could traverse before getting to line 28, there is now only one possible path that goes directly to line 28. Therefore, refactoring lines 28 to 33 makes that sequence testable without the control flow preceding it.

This suggestion in Listing 5.2 highlights one of the positives of this approach: that it can highlight straight-line code that should be moved into a separate function. On

the other hand, the reason described for generating the suggestion highlights one of the shortcomings. The current wording of this approach makes it seem removing these instructions decreases the paths of execution in `main`, which is incorrect. Removing these lines of code removes the paths of execution taken to get to this code - however it does not decrease the number of paths of execution in `main` given that this code is straight line code. This is one of the areas of improvement discussed in Chapter 7.

Each line in the program is compared as we compared line 28; the slice at each instruction is checked against the slice at each instruction with the variable group `inFile`. In the end this generates a list of instructions containing lines 28-33, all with a difference of 3 between the original slice and the reduced slice, as well as lines 38-45, 49-54, and a few other instructions not included in the code sample above.

The list of line numbers with a difference in the two slices is then expanded to include any line numbers containing unimportant instructions such as blank lines and comments as mentioned in Section 4.5. Therefore, prior to generating suggestions, the line numbers that suggestions are being generated over include lines containing comments (ex. 27, 35) and lines without code (e.g. 34, 46). This prevents formatting from affecting the suggestions.

Listing 5.3: Suggestion resulting from Listing 5.1.

```
1 line 38-45 (main):
2         parameters: inFile, pixels, rgb
3         returns: pixels
4         reason: Removing these instructions decreases number
5                 of paths of execution in the function -
6                 making the code more readable and testable.
```

Listing 5.4: Suggestion resulting from Listing 5.1.

```
1 line 49-54 (main):
2         parameters: height, i, picture, pixels, width
3         returns: picture
4         reason: Removing these instructions decreases number
5                 of paths of execution in the function -
6                 making the code more readable and testable.
```

When the suggestions are grouped, the first two suggestions end up as line 27-33, which we examined in Listing 5.3, and line 38-45, which is seen in Listing 5.4. The separation between the first and second suggestion is due to the fact that the complexity at line 37 and line 38 does not change regardless of the variable group. Both lines define an empty list meaning there are no reaching definitions for those instructions. The slice at those lines, regardless of the variable group, is a single block with a complexity of 1. This is also seen between the later two suggestions, which are separated by declarations to `picture` and `i` on lines 47 and 48.

Suggestions generated by *Earthworm* are often split on variable declarations. This is convenient, since variable declarations tend to indicate new functionality. In this case, the first suggestion group prints the header, the second suggestion group reads the pixels in the file, while the last suggestion group formats the pixels that were read in. All three tasks are very distinct.

Listing 5.5: Suggestion resulting from Listing 5.1.

```
1 line 41-45 (main):
2         parameters: comp, pixels, rgb
3         returns: pixels, rgb
4         reason: Removing these instructions decreases number
5                 of paths of execution in the function -
6                 making the code more readable and testable.
```

The suggestion shown in Listing 5.5 differs from the previously mentioned suggestions in that it was generated based on the variable group `rgb`. In Listing 5.1, a variable group of `rgb` decreased slice complexity by three or more for seven instructions: 41, 42, 43, 44, 45, 52, and 54. Given that line 52 and line 54 do not have any surrounding instructions that decrease in complexity, the only suggestion that is generated from removing `rgb` is instructions 41-45 which decrease from a slice complexity of 16 to a slice complexity between 10 and 13. This highlights a shortcoming where occasionally a suggestion can be over decomposed. Although it can be argued this suggestion does one thing - it handles tracking the pixel colors as they are coming in,

this section of code doesn't perform as clear of a task as the previous three that were identified.

The refactored code after using *Earthworm* on Listing 5.1 is provided in Appendix A.1.1. The code went from being a single function with 81 instructions (not including comments or empty lines) to eight functions with an average 12.5 instructions. The cyclomatic complexity went from 44 for the single function to an average of 6.125 for the eight functions.

5.1.1 Quality of Slow Flag

To give the user the option to select between faster performance and a more thorough analysis, *Earthworm* contains a 'slow' flag. The intention for adding the slow flag is to get the user engaged first by generating a shorter list of suggestions. After the user finds the tool useful, they are more likely to spend the extra time running the program using the slow flag to get more suggestions on how to improve their code. The slow flag only affects this method of generating suggestions because this approach is the most time consuming of the three analyses and has the most room for configuration.

When the slow flag is not used, only variable groups consisting of a single variable are generated. When the slow flag is used variable groups of size 1, 3, and 4 are generated. The effectiveness of the slow flag heavily depends upon the program being analyzed. For example, the slow flag has no affect on the program in Listing 5.1. This is in part due to the general simplicity and the limited number of intertwined variables within this code. In a case where there are more variables interacting together, the slow flag helps generate more quality suggestions. Occasionally, the slow flag generates too many suggestions.

The primary data set referenced during development consisted of student code

for 66 CPE 101 students. The program was a simplified ray casting program. The file on which *Earthworm* was running (named `cast.py`) included functions to calculate the color displayed in each pixel based on ray and sphere intersections. The functions in the program took in upwards of 10 parameters in addition to defining many local variables. An analysis of the data set shows that the slow flag took 12.1 minutes longer to run on all 66 files, averaging to 11 seconds per file, but produced 115 additionally suggestions overall, averaging out to 1.7 more suggestions per file. These new suggestions, however, were only generated for 26 out of the 66 input files. Similarly, the time increased significantly more on some files than others.

In general, the slow flag helps when more variables where individual variables are not useful in extracting code.

5.2 Similar References Consecutive Instructions

This section goes over example suggestions generated by the approach outlined in Section 4.3 which examines multiple consecutive instructions with the same local references.

The primary code sample used in this analysis is provided in Listing 5.6. The code sample is a part of a command line version of the 2048 game[2].

Listing 5.6: Example code for investigating consecutive similar references.

```
107 # Moves the board.
108 def move_board(game, direction):
109     has_shift = False
110     has_merge = False
111
112     if direction == 'w':
113         print('... UP ...')
114         game = transpose(game)
115         game, has_shift = shift_left(game)
116         game, has_merge = merge(game)
117         game, _ = shift_left(game)
```

[2]The full program is available in Appendix A.1.2

```
118         game = transpose(game)
119     elif direction == 'a':
120         print('... LEFT ...')
121         game, has_shift = shift_left(game)
122         game, has_merge = merge(game)
123         game, _ = shift_left(game)
124     elif direction == 's':
125         print('... RIGHT ...')
126         game = reverse(game)
127         game, has_shift = shift_left(game)
128         game, has_merge = merge(game)
129         game, _ = shift_left(game)
130         game = reverse(game)
131     elif direction == 'z':
132         print('... DOWN ...')
133         game = reverse(transpose(game))
134         game, has_shift = shift_left(game)
135         game, has_merge = merge(game)
136         game, _ = shift_left(game)
137         game = transpose(reverse(game))
138     else:
139         print('... INVALID MOVE ...')
140 
141     made_move = has_shift or has_merge
142     return (game, made_move)
```

Four suggestions are generated on this subsection of the program, as shown in Listing 5.7, Listing 5.8, Listing 5.9, and Listing 5.10. Each suggestion is generated because the only referenced variable in each suggestion's group is `game`. The suggestions are separated because the conditionals between the suggestion groups only reference the variable `direction` and the print statements in between the suggestions have no local references (`print` is a non-local reference so it is not considered for this analysis).

Listing 5.7: Suggestion resulting from Listing 5.6.

```
1 line 114-118 (move_board):
2         parameters: game
3         returns: game, has_merge, has_shift
4         reason: The same set of variables are referenced in
5                 all instructions in the given line numbers.
```

Listing 5.8: Suggestion resulting from Listing 5.6.

```
line 121-123 (move_board):
        parameters: game
        returns: game, has_merge, has_shift
        reason: The same set of variables are referenced in
                all instructions in the given line numbers.
```

Listing 5.9: Suggestion resulting from Listing 5.6.

```
line 126-130 (move_board):
        parameters: game
        returns: game, has_merge, has_shift
        reason: The same set of variables are referenced in
                all instructions in the given line numbers.
```

Listing 5.10: Suggestion resulting from Listing 5.6.

```
line 133-137 (move_board):
        parameters: game
        returns: game, has_merge, has_shift
        reason: The same set of variables are referenced in
                all instructions in the given line numbers.
```

The listings above highlight one of the issues with *Earthworm* that is discussed in more depth over the next two chapters. Often times, when the original code is repetitive, the suggestions become repetitive. Though these suggestions are considered to be of quality, such repetition can overwhelm a beginning programmer. This primarily becomes an issue when it leads to multiple functions performing relatively similar tasks. Detecting such code clones is outside the scope of this analysis.

The Similar References Consecutive Instructions technique also commonly generates suggestions when a large set of referenced variables is used to compute three related values, such as a point's x-position, y-position, and z-position. This scenario typically happens over three lines of code because suggestions need to have at minimum three lines of code to be output and it is less common for four or more instructions to be that deeply related. This was commonly seen in `cast.py` when the code defined the red, green, and blue values for the color at a given pixel. An

example is provided in Listing 5.11 where the variables `N`, `ldir`, `light`, and `sphere` were all used in consecutive instructions.

Listing 5.11: Lines from a cast.py program showing similar references approach.

```
90      r = dot_vector(N, ldir)*light.color.r*sphere.color.r*sphere.finish.
            diffuse
91      g = dot_vector(N, ldir)*light.color.g*sphere.color.g*sphere.finish.
            diffuse
92      b = dot_vector(N, ldir)*light.color.b*sphere.color.b*sphere.finish.
            diffuse
```

The refactored code after using *Earthworm* on Listing 5.6 is provided in Appendix A.1.2. The code went from being 10 functions with an average 12.6 instructions (not including comments or empty lines) to 17 functions with an average 8.2 instructions. The cyclomatic complexity went from an average 8.6 per function prior to refactoring to an average of 5.5 per function after refactoring.

5.3 Differences in Live Variables and Referenced Variables

This section goes over example suggestions generated by the approach outlined in Section 4.4 which finds suggestions based on differences in the referenced set and live variables of a block or instruction. Recall the intention of this approach is to find suggestions where the variables are passing through the code such that they are defined before and used after.

One example of this analysis performed at the block level can be seen in Listing 5.13. Listing 5.13 shows a suggestion for the `cast_ray` function in Listing 5.12. The suggestion is generated because the live variable set passing through the instructions on lines 20-25 contains four more variables than the largest reference set in the blocks containing those instructions. For example, at the block containing line 20, the referenced set is empty while the live variable set contains {`'hits'`, `'color'`, `'sphere_list'`, `'ray'`, `'eye'`, `'light'`}. The block containing lines 22 to 24 has

a reference set of {‘i’, ‘hits’, ‘ray’, ‘small’} and a live variable set containing 10 variables. By default, the minimum difference required between live variables and referenced sets of a block to generate a suggestion is four.

Listing 5.12: Example code for investigating difference in live variables and reference variables at the block level.

```
15 def cast_ray(ray, sphere_list, color, light, eye):
16     hits = col.find_intersection_points(sphere_list, ray)
17     if (len(hits) == 0):
18         return data.Color(1.0, 1.0, 1.0)
19     else:
20         small = 0
21         for i in range(1, len(hits)):
22             cur = vm.length_vector(vm.difference_point(ray.pt, hits[i
                   ][1]))
23             smallest = vm.length_vector(vm.difference_point(ray.pt, hits
                   [small][1]))
24             if cur < smallest:
25                 small = i
26
27         ambientColor = compute_ambient_lighting(hits[small][0], color)
28         pointLighting = compute_point_and_specular_light(hits[small][1],
                hits[small][0], light, sphere_list, eye)
29
30         return color_add(ambientColor, pointLighting)
```

Listing 5.13: Suggestion resulting from Listing 5.12

```
1 line 20-25 (cast_ray):
2     parameters: hits, ray
3     returns: small
4     reason: Multiple variables defined prior to these
5             instructions are not used in these line numbers.
```

Although the block and instruction level analysis often generate the same results, such as in the prior example, there are enough differences between the suggestions generated to include both in the final version of `Earthworm`. Listing 5.14 examines one example suggestion which is only generated by the instruction level analysis[3]. The suggestion itself can be seen in Listing 5.15. Similar to the block level analysis, by default, the minimum difference required between the live variable set and the

[3]Some instructions in Listing 5.14 have been replaced with ellipses.

referenced set of an instruction to generate a suggestion is four.

Listing 5.14: Example code for investigating difference in live variables and reference variables at the instruction level.

```
6  def cast_ray(ray, sphere_list, color, light, point):
7      listing = collisions.find_intersection_point(sphere_list, ray)
8      if listing == []:
9          return data.Color(1.0,1.0,1.0)
10     else:
11         mind = 0
12         for i in range(1,len(listing)):
13             a = vector_math.length_vector(vector_math.difference_vector(
                   ray.pt, listing[i][1]))
14             b = vector_math.length_vector(vector_math.difference_vector(
                   ray.pt, listing[mind][1]))
15             if a < b:
16                 mind = i
17         diffuse_value = visibility(listing[mind][0], listing[mind][1],
               light, sphere_list)
18         specular_value = specular(ray, listing[mind][0], listing[mind][1],
               light, sphere_list)
19         spheres = listing[mind][0]
20         red = ...
21         green = ...
22         blue = ...
23         Sphere_color = data.Color(red, green, blue)
24         return Sphere_color
```

Walking through how the suggestion was generated, first the difference in live variable set and reference set are calculated. In `cast_ray` lines 8, 10, 11, 12, 13, 14, 15, and 16 all had live variable sets containing four variables more than their referenced sets. The final suggestion only contains lines 11-16 due to the methods of grouping instructions discussed in Section 4.5.2. In particular, instructions 8 and 10 need to remain together because they are multiline instructions. In addition, instruction 8 requires all parts of the `if` body (in this case, line 9) to be included in the suggestion. Similarly, instruction 10 requires all parts of the `else` body (in this case, lines 11-24) to be included in the suggestion. Therefore, the final suggestion as seen in Listing 5.15 consists of lines 11-16 since neither instruction 9 nor the instructions after 16 are included in the original list of instructions. This suggestion is not generated by

the block level analysis because the block containing instructions 13, 14, and 15 had only three more variables in the live variables set than in the referenced set.

Listing 5.15: Suggestion resulting from Listing 5.14

```
1 line 11-16 (cast_ray):
2    parameters: listing, ray
3    returns: mind
4    reason: Multiple variables defined prior to these
5            instructions are not used in these line numbers.
```

5.4 Missed Suggestions

Even with multiple approaches and the use of the slow flag, some good functional decomposition examples are missed. Additionally, some of the suggestions generated, in particular for the last generation method, do not improve the code decomposition.

An example of a suggestion that is of questionable value can be seen in Listing 5.17 for the `cast_all_rays` function in Listing 5.16. The suggestion is generated based on the difference in the live variables and referenced sets for the instructions on line 33 through 36. However, the suggestion does not perform an identifiable task that is disjoint from the current function it is in.

Listing 5.16: Example code for investigating difference in live variables and reference variables at the instruction level which resulted in a poor suggestion

```
20 def cast_all_rays(view, eye_point, sphere_list, ambient, light):
21     # min_x, max_x, min_y, max_y, width, height
22     current_x = view.min_x
23     current_y = view.max_y
24     xstep = (view.max_x - view.min_x) / float(view.width)
25     ystep = (view.min_y - view.max_y) / float(view.height)
26     pixels = []
27     while current_y > view.min_y:
28         while current_x < view.max_x:
29             ray = Ray(eye_point, vector_from_to(eye_point, Point(
                   current_x, current_y, 0)))
30             color = cast_ray(ray, sphere_list, ambient,  light)
31             pixels.append(get_color_string(color))
32             current_x += xstep
33         current_y += ystep
34         current_x = view.min_x
35         if current_y % 10 == 0:
36             print "PART DONE!"
37     print_p3(view.width, view.height, pixels)
```

Listing 5.17: Suggestion resulting from Listing 5.16

```
1 line 33-36 (cast_all_rays):
2         parameters: current_y, view, ystep
3         returns: current_x, current_y
4         reason: Multiple variables defined prior to these
5              instructions are not used in these line numbers.
```

One example of missing decomposition can be seen in Listing 5.1. As previously mentioned, in the Listing 5.1, lines 8 to 26 are composed of a conditional nested within an exception handling block. This segment of code (whose full code can be found in Appendix A.1.1) only references the previously defined variable `file_name` and returns the variable `inFile`. It performs a single task - opening the file with the given file name or exiting if it is unable to do so. All factors considered, lines 8 to 26 make a strong case for being identified as a suggestion.

Unfortunately, recall that *Earthworm* fails to generate a suggestion on this exception handling code. Walking through each suggestion generation method, it can be

understood why this happens, demonstrating a shortcoming of *Earthworm*. The first generation method of investigating slices fails because there is no code complexity prior to this segment. The only referenced variable was `file_name` and removing it makes no difference to code complexity. The next generation method looks at common references across multiple lines which is not intended to target complex control flow. The last generation method examines differences in the live variables set and the referenced variables set. The only references defined prior to line 8 are `file_name` and `outFile`. Therefore, *Earthworm* fails to generate a suggestion on these lines.

It is also important to note, if the user were to decompose lines 8 to 26 into a function prior to decomposing the suggestions in Listing 5.2, Listing 5.3, and Listing 5.4, the aforementioned listings would not longer appear. Making `inFile` a parameter would no longer decrease each code block's complexity if the code were to be decomposed into a function. Therefore, either a suggestion generated on the exception handling block would need to be presented after the user decomposes the aforementioned listings, or the generation method that identifies lines 8 to 26 as a suggestion would also need to identify the listings (since most users rerun the program after every change due to the line numbers changing). Given that the tool itself does not refactor the code, just presenting the suggestion from lines 8 to 26 would not prevent the other listings from being output; the suggestions would only stop appearing if the user were to extract lines 8 to 26 before running the tool.

5.5 Linter

The code sample in Listing 5.18 demonstrates examples of issues with student code that are identified by *Earthworm*'s linter. The suggestions output are given in Listing 5.19.

Listing 5.18: Example code for investing Earthworm's linter.

```
3  def␣has_greater_than_n(numbers,␣n):
4  ␣␣result␣=␣False
5  ␣␣for␣num␣in␣numbers:
6  ␣␣␣␣if␣num␣>␣n:
7  ␣␣␣␣␣␣result␣=␣True
8  ␣␣if␣result␣==␣True:
9  ␣␣␣␣return␣True
10 ␣␣else:
11 ␣␣␣␣return␣False
12
13 def␣get_in_range(numbers,␣min_number,␣max_number):
14 ␣␣␣␣␣return␣[number␣for␣number␣in␣numbers␣if␣number␣>=␣min_number␣and␣
       number␣<=␣max_number]
```

Listing 5.19: Suggestions resulting from Listing 5.18.

```
1 line 3: Indentation should be either 3 or 4 spaces.
2         Use descriptive variable name instead of
3         'n' in 'has_greater_than_n'.
4 line 8: Rewrite conditional as a single line return
5         statement: 'return <conditional>'.
6 line 13: 'get_in_range' has different indentation than
7         'has_greater_than_n'.
8 line 14: Line length over 80 characters.
```

Walking through the suggestions, the first one indicates the indentation should be either three or four spaces to conform to local recommendations based on PEP 8 style guide (which requires four spaces). This suggestion is a top level suggestion that only occurs once through the program - even if other functions fail to meet this requirement. Additionally, the function contains a variable `n` that contains only a single letter. The suggestion on line 8 tells students to simplify the conditional on that line into a single line. The suggestion on line 13 once again identifies differences in indentation. This is a function level suggestion to prevent too many repeated errors of this sort. Lastly, the suggestion on line 14 occurs for every line with length over 80 characters.

Chapter 6

VALIDATION

This chapter outlines the two approaches used to validate *Earthworm*: a student lab completed by introductory computer science students and a survey for individuals familiar with computer science. Although both approaches included surveys[1] taken primarily by students, in this chapter the first will be referenced as the *student survey* and the second will be referred to as the *expert survey*.

The student survey was run as a 50-minute lab for students in CPE 101. The lab[2] had the students run *Earthworm* on two Python source code files they had not previously seen[3]. The students were asked to implement the changes suggested by *Earthworm* to improve the programs' functional decomposition. Additionally, each student filled out two surveys - one prior to using *Earthworm* and one after. In total, 32 students in one section of CPE 101 completed the lab and related surveys.

The expert survey was aimed at individuals, referred to as experts for the remainder of this chapter, who had previously taken 3+ college courses in computer science, had 1+ years of experience in industry, or had been an instructor for computer science. The survey consisted of evaluating the suggestions generated for three code samples; the code samples and their corresponding suggestions are included in Appendix A.1.1 (referred to as Code Sample 1 in the following sections), Appendix A.1.2 (referred to as Code Sample 2), and Appendix A.1.3 (referred to as Code Sample 3). Although it would have been ideal to have more code samples on which to perform additional analysis, each code sample took 10-15 minutes to respond to making the current survey take between 30-45 minutes.

[1] All surveys are available at *Earthworm*'s github repository in the folder `paper/surveys` [2].
[2] The lab is available at *Earthworm*'s github repository in the folder `paper/lab` [2].
[3] The source code is included in Appendix A.1.3 and Appendix A.1.4.

For each code sample, the experts reviewed the text and corresponding code for each suggestion individually on overall usefulness, helpfulness of the text, and improvement in ability to unit test the code. At the end of each code sample, the experts were asked to provide feedback on the overall improvement in the decomposition of the program. The survey results below analyze results from the 31 individuals who responded to the survey. The demographic was primarily current and previous Cal Poly students mixed in with a handful of non-Cal Poly students. The survey was promoted through Facebook groups internal and external to Cal Poly.

The following sections analyze *Earthworm*'s performance on three metrics: overall usefulness, helpfulness of text, and improvement in the ability to unit test the code. Each metric was scored on a scale of 1 to 7 where 1 indicated *Not at all* and 7 indicated *Very useful/helpful*. The corresponding percentage for a given score out of 7 was calculated using Equation 6.1:

$$percentage = (score - 1.0)/6.0 * 100 \tag{6.1}$$

6.1 Student Survey: General Feedback

The student survey hoped to identify if students felt more confident in their abilities to decompose code after using the tool and if they found the tool easy to use and to be overall useful.

Students were asked to evaluate themselves on how they would rate themselves at code decomposition in both the pre-survey and the final survey. The average score on a scale of 1 to 7 went from 3.5 (or 41.7% based on Equation 6.1) in the pre-survey to 3.53 (or 42.2%) in the final survey.

Although the average appeared to show no difference in comprehension, the distribution of scores changed significantly as seen in Figure 6.1. After 40 minutes of

using the tool the scores shift from an aggregation around 3 to a distinct peak at 5. The fact that the average did not change despite the change in distribution might suggest that some students initially overestimated their decomposition abilities and could gain insight from using such a tool.

Interestingly, the increase in the number of students who rated their decomposition skills as 1 out of 7 appeared to correlate with those who had difficulty using the tool. This theory is supported by Table 6.1 which shows the number of students who had specific differences between the tool's ease of use and how they would score themselves on decomposition. Of the 32 students, 75% of the students rated their skills to be within a point of their score for ease of use. I believe this shows strong evidence of a correlation between using *Earthworm* and confidence in decomposing code.

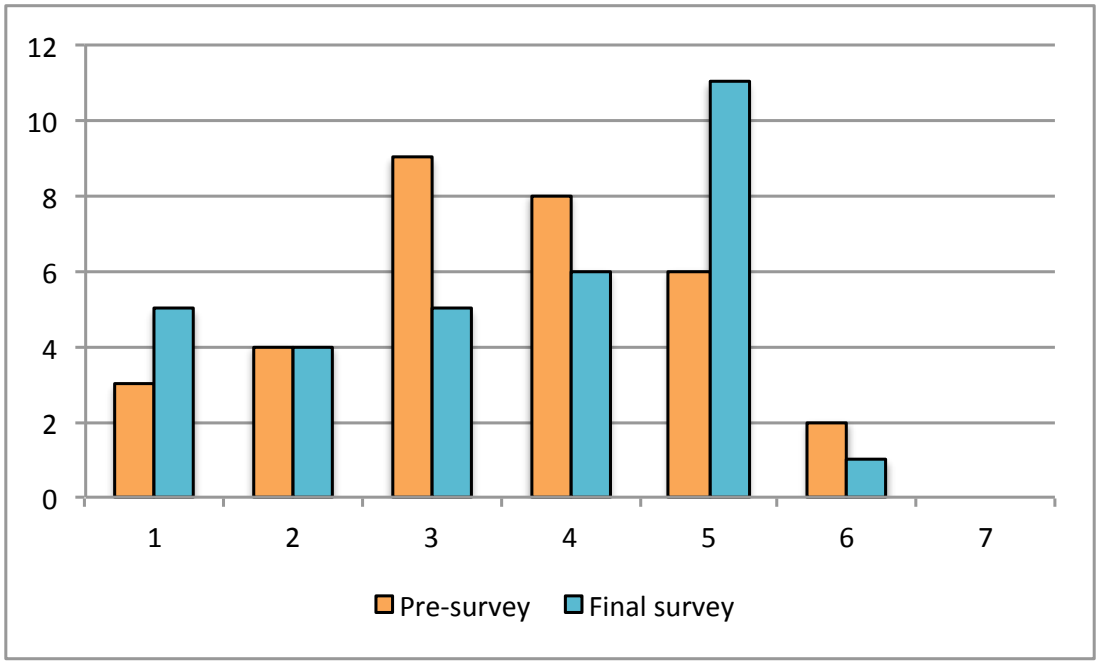

Figure 6.1: Histogram of self evaluation from pre-survey to final survey.

In addition to evaluating ease of use, students evaluated *Earthworm* on how easy it was to understand what they should be changing in the code. Both criteria had scores averaging around 4.5. The distribution of scores from 1 to 7 along with the self-evaluation scores can be seen in Figure 6.2. The y-axis contains the count of a

Table 6.1: Table showing difference in student self evaluation scores versus ease of use scores

Difference (ease of use score - self evaluation score)	Count	Percentage of class
-2	1	3.1%
-1	3	9.4%
0	12	37.5%
1	9	28.1%
2	5	15.6%
3	1	3.1%
4	0	0%
5	1	3.1%

particular score while the score is plotted on the x-axis. This graph further emphasizes the clustering of scores in similar patterns - suggesting a relationship between the tool and the self-evaluation score for decomposition.

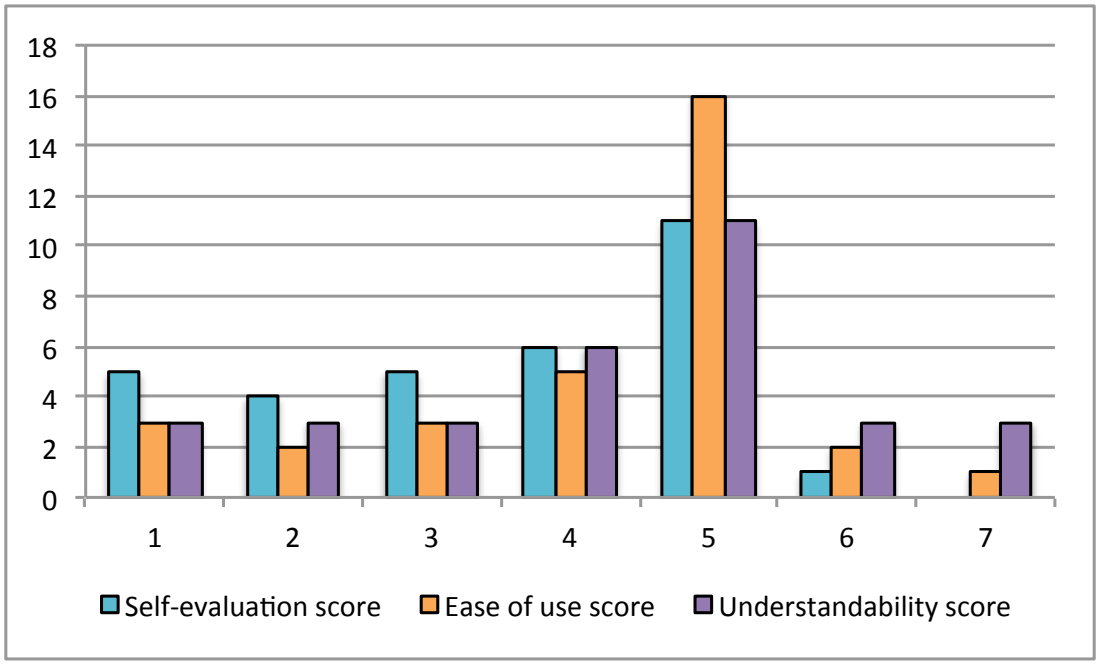

Figure 6.2: Histogram summarizing the student's final survey results.

All in all, these metrics suggest *Earthworm* would be useful in teaching and improving student's decomposition assuming the suggestions are of high quality. The following two sections evaluate the quality of the suggestions generated by *Earthworm* from the expert's point of view.

6.2 Expert Survey: Overall Usefulness of Suggestions

This section analyzes the expert survey results for overall usefulness of the suggestions provided by `Earthworm`.

6.2.1 Overall Usefulness of All Suggestions

The graphs below show the reported overall usefulness of the suggestions generated by *Earthworm* for each code sample. The responses regarding the suggestions for Code Sample 1, Code Sample 2, and Code Sample 3 are summarized in Figure 6.3, Figure 6.4, and Figure 6.5 respectively[4].

The y-axis of the plots contains the scores from 1 to 7. The x-axis contains the suggestion number. Each box-plot depicts the first quartile, median, and third quartile of the aggregation of the 31 users' usefulness scores for each suggestion. The white diamonds connected by the solid line plot the mean usefulness score for the given suggestion. For each chart, the right most box-plot shows the usefulness scores for all suggestions in that code sample averaged together.

When looking at the plots, the means and the medians suggest relatively positive impressions. However, there was variation by suggestion; some suggestions fared significantly better than others. For most suggestions there wasn't a huge spread of scores from the first through third quartile, although the minimum value was considerably lower.

[4]The full programs are available in in Appendix A.

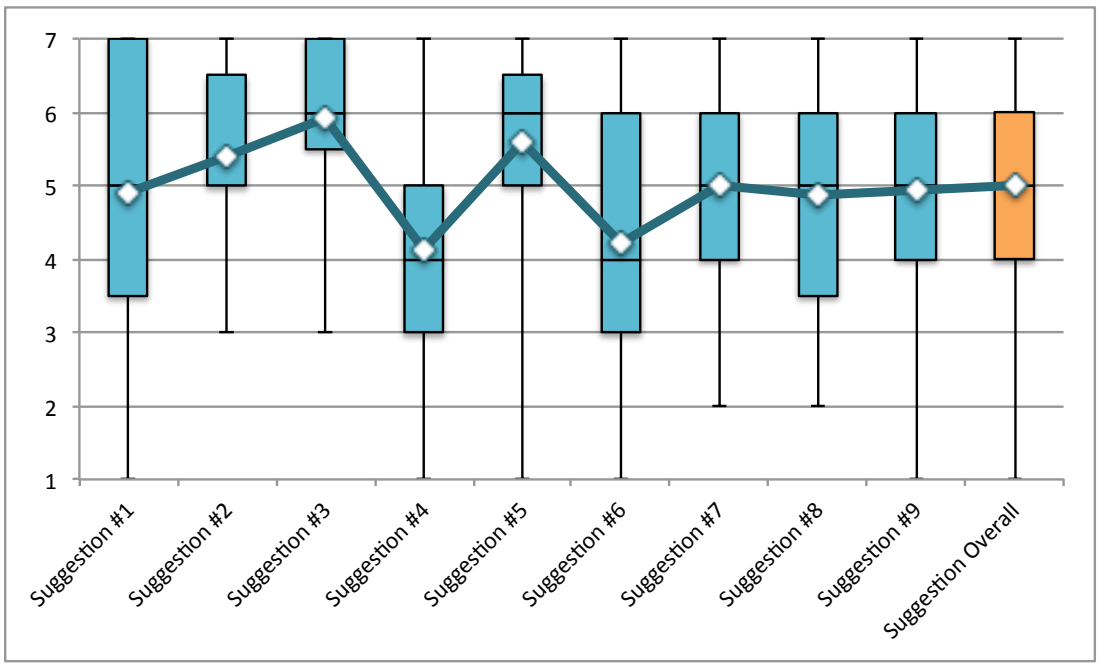

Figure 6.3: Box plot of average usefulness of suggestions generated on Code Sample 1.

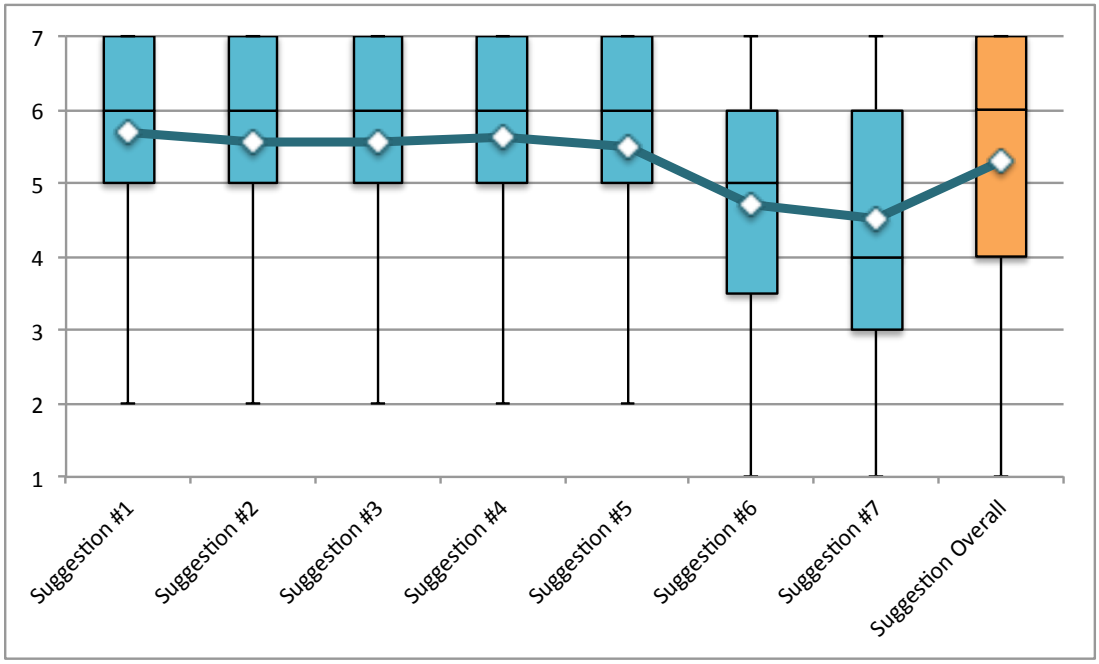

Figure 6.4: Box plot of average usefulness of suggestions generated on Code Sample 2.

The highest mean usefulness score in Code Sample 1 is for Suggestion #3, which can be seen in Listing 6.1. The first four suggestions of Code Sample 2 fared similarly,

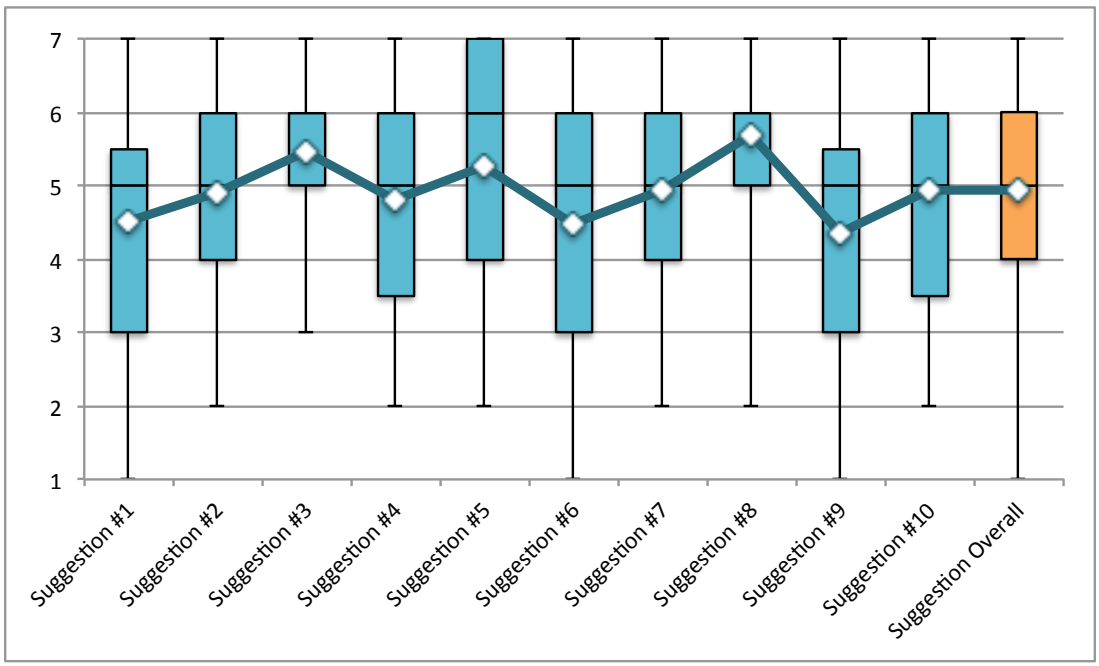

Figure 6.5: Box plot of average usefulness of suggestions generated on Code Sample 3.

although they did not have as high of a mean score. Although there is no clear distinction as to why Suggestion #3 in Code Sample 1 fared so well, I speculate it could be for a few reasons. First off, it was the first code segment in the survey to define a very distinct task. It is clearly visible this piece of code is intended to read the pixels from the file just by reading lines 35 through 37. The reasons for refactoring the lines listed in the prior two suggestions from Code Sample 1 were less clear. Additionally, the segment listed in Suggestion #3 contains a conditional inside of nested `for` loops and therefore is very clearly something that might add complexity to the program.

Listing 6.1: Highest rated code sample

```
35 # Read file.
36 pixels = []
37 rgb = []
38 for line in inFile:
39     line = line.split()
40     for comp in line:
41         if len(rgb) != 3:
42             rgb.append(comp)
43         if len(rgb) == 3:
44             pixels.append(rgb)
45             rgb = []
```

The usefulness data shown in the plots is brought together in Table 6.2, which lists the average usefulness, scored out of 7 for each code sample. For all three code samples combined, the average usefulness was rated as 5.05 out of 7 - or 67.6%.

Given that the average usefulness score for each code sample is within 6% of each other - it can be assumed the 67.4% is indicative of how well *Earthworm* might perform on other code samples. This score indicates there is still significant room for improvement, but that there is value in using *Earthworm* as it stands. Section 6.2.3 breaks down the usefulness of suggestions by the suggestion type, giving insight into the suggestion types that would benefit from refinement.

Table 6.2: Average usefulness of suggestions

Code Sample	Score (out of 7)	Score (as percentage)
Code Sample 1	5.0	66.6%
Code Sample 2	5.3	71.7%
Code Sample 3	4.93	65.5%
Overall	5.05	67.4%

6.2.2 Overall Usefulness by User Experience

To get a better understanding of why experts voted as they did, I tried to group the experts into categories. Given the limited data set, the most viable metric that separated the experts was the number of internships. The analysis below splits the experts into two groups: those with 0-1 internships who are considered the less experienced experts and those with 2+ internships who are considered the more experienced experts. The intention of this analysis is to see if more industry experience indicates any distinguishable difference in how users feel about suggestions generated by *Earthworm*. Out of the 31 people who completed the expert survey, 12 had 0-1 internships while 19 had 2+ internships.

Figure 6.6 contains box-plots showing the difference in usefulness according to individuals with more or less industry experience. As before, the y-axis contains the possible scores from 1 to 7. Individuals with 2+ internships had a slightly lower usefulness average than individuals with 0-1 internships. However, there is not enough variability to draw significant conclusions.

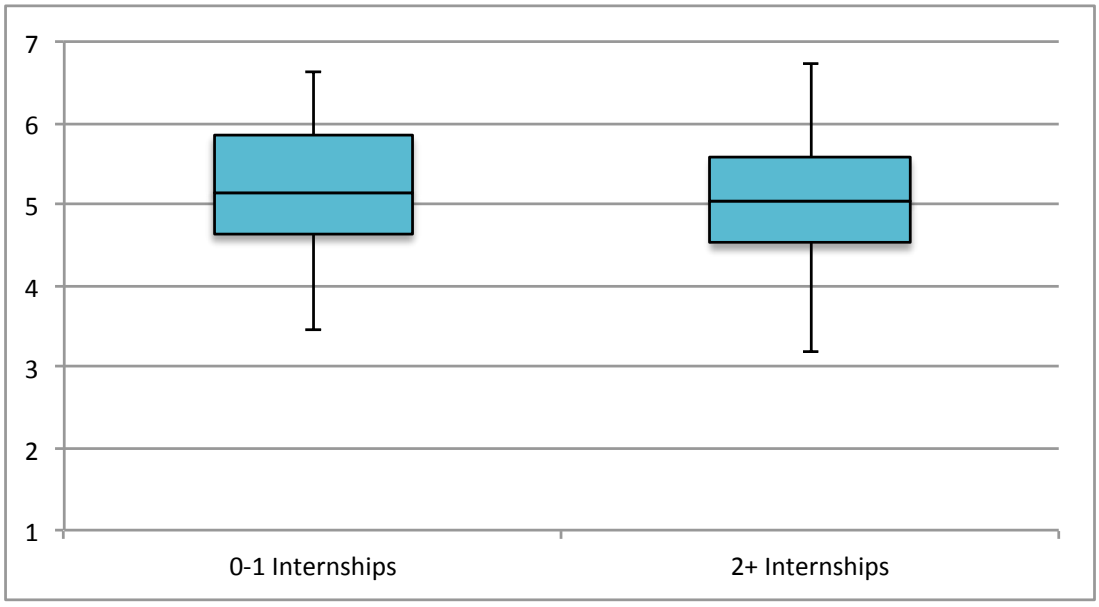

Figure 6.6: Box-plot of average usefulness of suggestions divided by experience.

Breaking this down further shows a more interesting result. The histogram in Figure 6.7 shows the average occurrence of a given score for an individual based on the number of internships. The y-axis indicates the number of suggestions and the x-axis indicates the score from 1 to 7. For example, the left bar for x-axis number 6 indicates that on average someone with 0-1 internships ranked 7 suggestions with a score of 6.

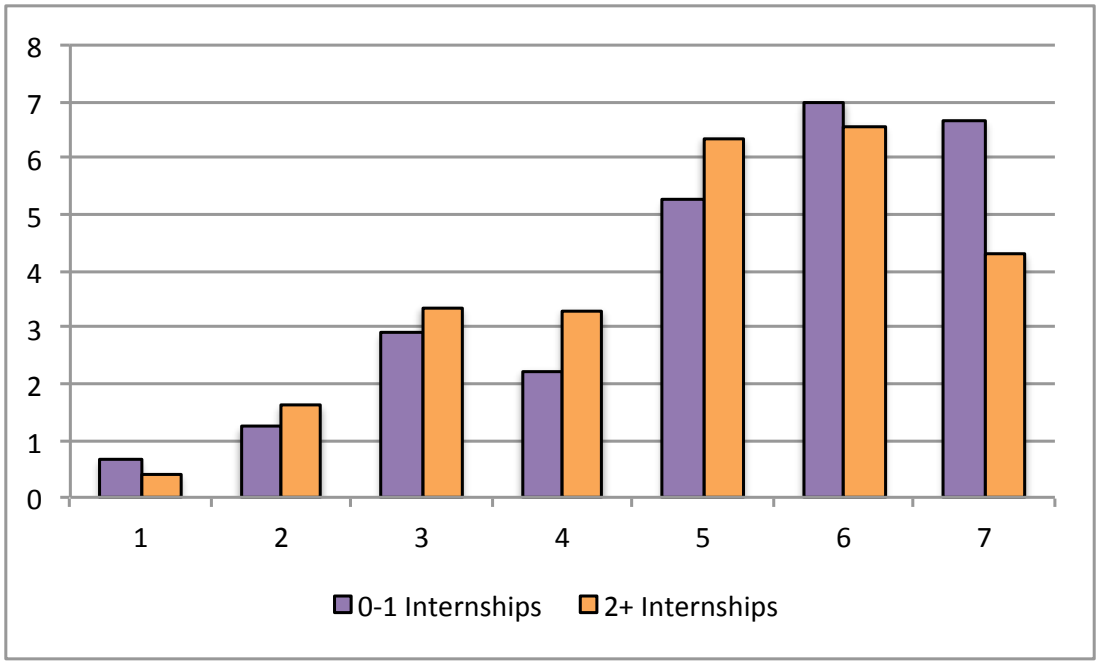

Figure 6.7: Histogram of average occurrence of scores 1 through 7 divided by experience.

The more experienced individuals gave slightly lower scores, giving the most scores between the range 5 to 6. Whereas non-experienced individuals gave the majority of scores of either 6 or 7. One potential reason is that people with fewer internships likely had seen less code, and therefore had less prior experience to determine whether a suggestion was good or not. For those with fewer internships, I presume the suggestion was good if it appeared to take out segments of the code that appeared to be doing one thing. However, I assume the more experienced individuals were able to identify the more subtle flaws in the automated analysis which are discussed in Section 6.5.

The graph in Figure 6.8, plotting the percentage of suggestions that were easier to test according to internship experience, reiterated the previous sentiment. Overall, those with more experience were more critical of the suggestions. However, even those with 2+ internships had a median of 81%; meaning 81% of the suggestions would make unit testing easier.

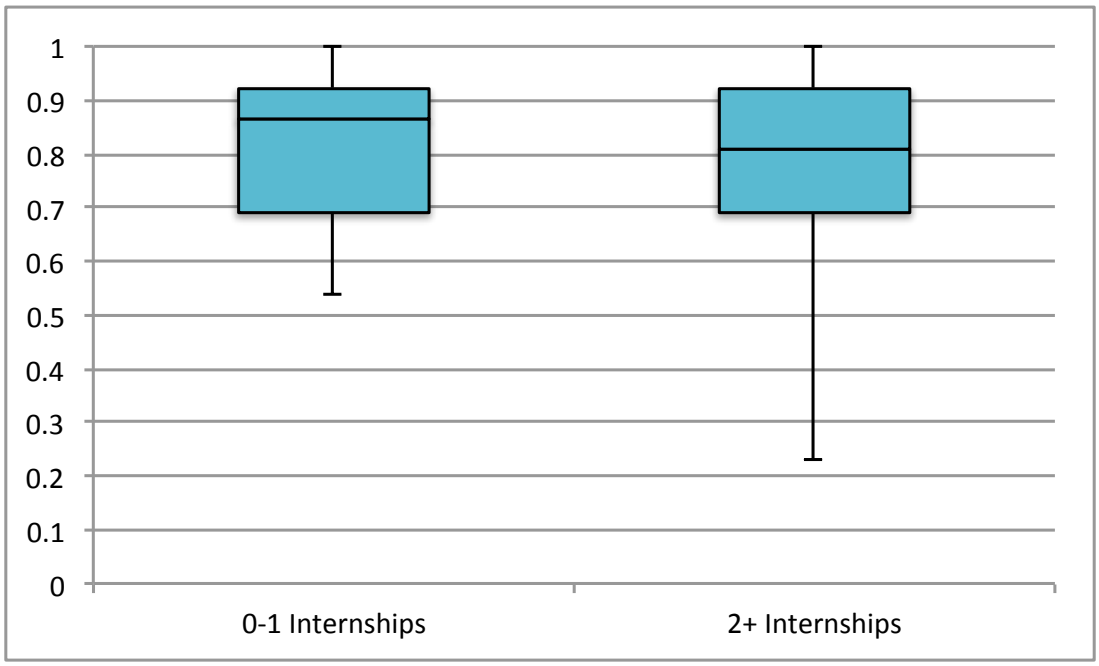

Figure 6.8: Box-plot of percentage of suggestions that are easier to test divided by experience.

6.2.3 Overall Usefulness by suggestion type

The visualizations in this section demonstrate the usefulness of a suggestion based on suggestion type, or generation method. This was intended to determine which generation methods discussed in Chapter 4 were most effective. For the remainder of this chapter, the suggestions generated using slices as described in Section 4.2 are considered Suggestion Type 1. The suggestions generated using redundant references as described in Section 4.3 are considered Suggestion Type 2. Lastly, the suggestions generated from the differences in live variables and referenced variables discussed

in Section 4.4 are considered Suggestion Type 3. In total, 16 suggestions were of Suggestion Type 1, 6 suggestions were of Suggestion Type 2, and 4 suggestions were of Suggestion Type 3.

Table 6.3: Average usefulness of suggestion by suggestion type

suggestion type	Score (out of 7)	Score (as percentage)
Type 1	5.03	67.2%
Type 2	5.43	73.8%
Type 3	4.45	57.5%

Table 6.3 shows the difference in how well each suggestion type was received. Suggestion Type 3 performed worse than the other approaches. All four suggestions generated from that approach had an average usefulness score between 4.3 to 4.9 (or 56% to 65%).

Meanwhile, Suggestion Type 2 ranked the highest with an average score of all suggestions of that type being 5.43 - or nearly 74%. Although it is not possible to know exactly why it ranked so high, I believe this is because Suggestion Type 2 is the easiest to visualize. The suggestion is generated when the same reference set is used across multiple consecutive instructions, which is something that is relatively easy to see and identify. Additionally, given that this approach is very specific, it causes Suggestion Type 2 to identify very similar issues in the code which should likely receive nearly identical scores. In this case all of the suggestions of Suggestion Type 2 came from Code Sample 2; where the first four suggestions were nearly identical.

The most used suggestion group of the three was Suggestion Type 1 with 16 of the 26 suggestions. It had the most variability in scores but overall performed well - getting an average score of 5.03 (or 67.2%). I believe one of the challenges in getting a higher score was the complexity of the generation method which often made the reasoning for the suggestion less intuitive. As such, it became more difficult for the

user to visualize the value of pulling out the code in certain scenarios.

6.2.4 Overall Usefulness by Suggestion Size

The visualizations in this section demonstrate the usefulness of a suggestion based on the suggestion size. Figure 6.9 is a scatter plot of the size of the suggestion, on the x-axis, versus the score of the suggestion, on the y-axis. Figure 6.10 is a similar graph highlighting the suggestions with fewer than 20 lines. The scatter plots indicate there is no immediate correlation between the number of instructions in a suggestion and how useful the suggestion is. Due to the variability in the size of the suggestions, this is one area of analysis that needs more data points in order to draw any meaningful conclusions.

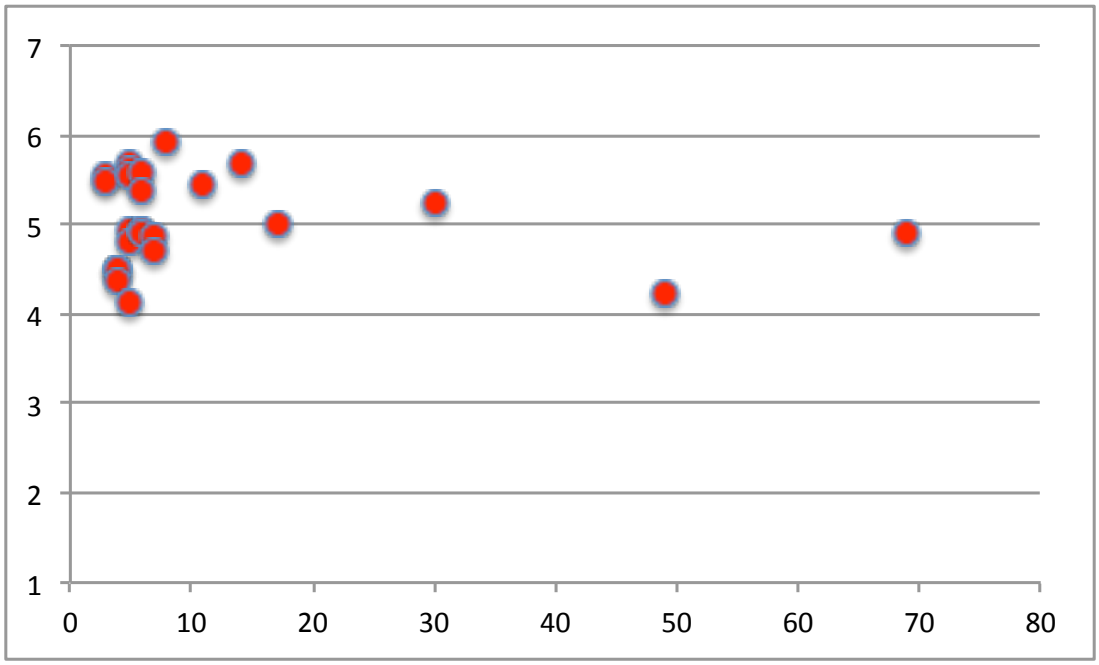

Figure 6.9: Scatter plot showing usefulness of suggestions by size.

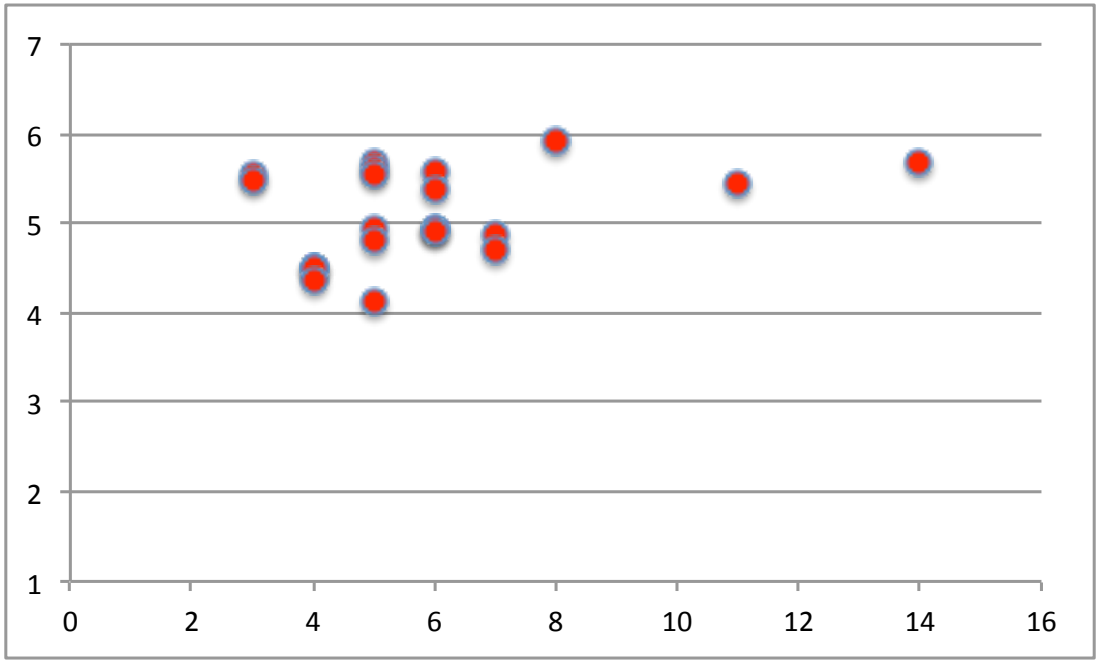

Figure 6.10: Scatter plot showing usefulness of suggestions by size for suggestions less than 20 lines.

6.3 Expert Survey: Overall Helpfulness of Text

This section discusses the perceived helpfulness of the text of the suggestions provided by `Earthworm` (indicating line, parameters, and results for a proposed new function) based on the responses in the expert survey.

6.3.1 Helpfulness of All Suggestions

The graphs below show the helpfulness of the text of the suggestions for each code sample included on the expert survey. The responses to the suggestions for Code Sample 1, Code Sample 2, and Code Sample 3 are displayed in Figure 6.11, Figure 6.12, and 6.13 respectively[5].

The y-axes of the plots contain the scores from 1 to 7 and the x-axes contain the suggestion number. Each box-plot depicts the first quartile, median, and third

[5]The full programs are available in in Appendix A.

quartile of the aggregation of the 31 users' helpfulness scores for each suggestion. The white diamonds connected by the solid line plot the mean helpfulness score for the given suggestion. For each chart, the right most box-plot or bar shows the helpfulness scores for all suggestions in that code sample averaged together.

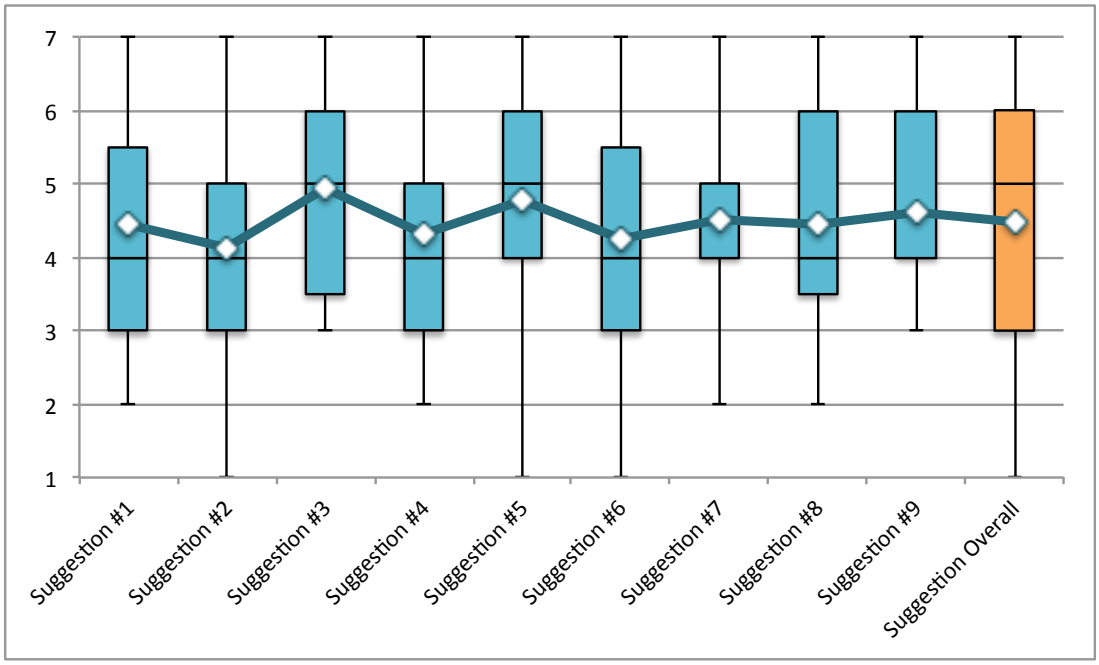

Figure 6.11: Box plot of average helpfulness of text for for suggestions generated on Code Sample 1.

This data is brought together in Table 6.4, which lists the average helpfulness of the suggestion text, scored out of 7 for each code sample. For all three code samples combined, the average helpfulness was rated as 4.71 out of 7 - or 61.8%.

This score was roughly 6% lower than usefulness indicating there is room for even more improvement in this category. Section 6.5 discusses the feedback provided by both students and experts on improving the text's helpfulness.

6.3.2 Helpfulness by User Experience

Similar to Section 6.2.2, this subsection examines the helpfulness of the text based on the industry experience of the user filling out the expert survey. To reiterate there

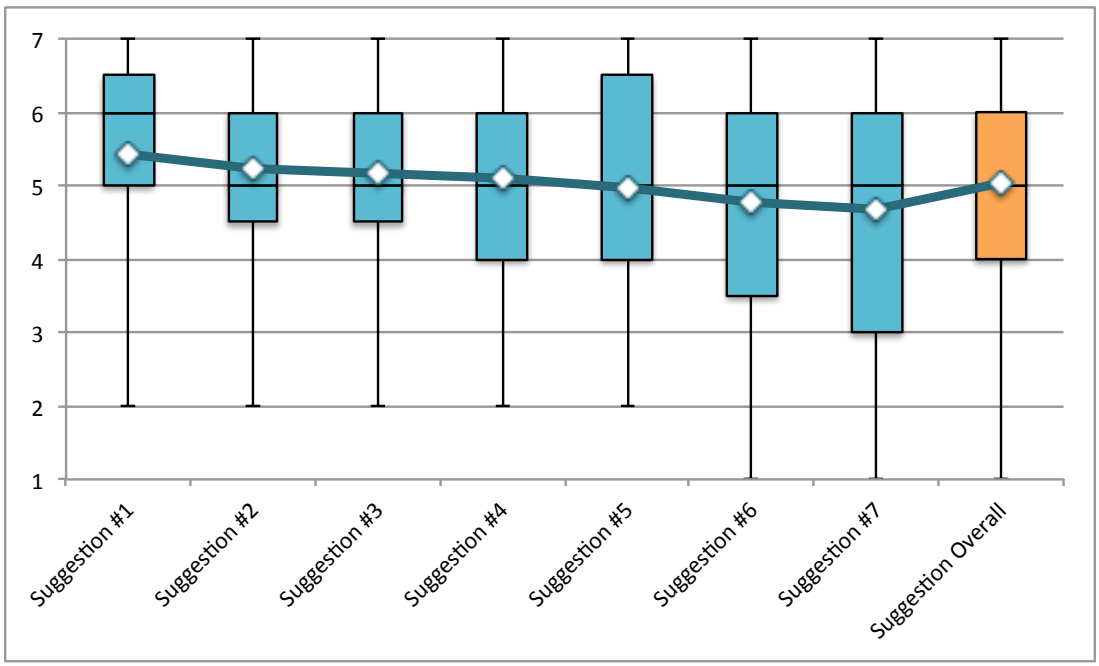

Figure 6.12: Box plot of average helpfulness of text for suggestions generated on Code Sample 2.

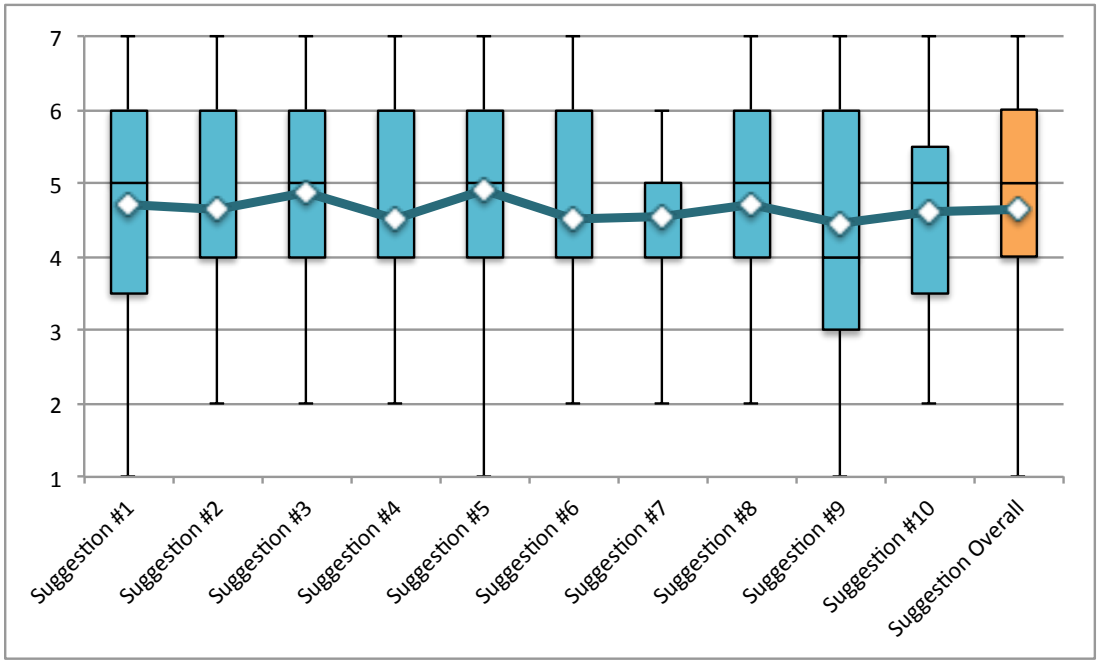

Figure 6.13: Box plot of average helpfulness of text for suggestions generated on Code Sample 3.

were 12 individuals with 0-1 internships and 19 individuals with 2+ internships.

Figure 6.14 shows those with more internship experience tended to have similar,

Table 6.4: Average helpfulness of text.

Code Sample	Score (out of 7)	Score (as percentage)
Code Sample 1	4.49	58.2%
Code Sample 2	5.04	67.4%
Code Sample 3	4.64	60.8%
Overall	4.71	61.8%

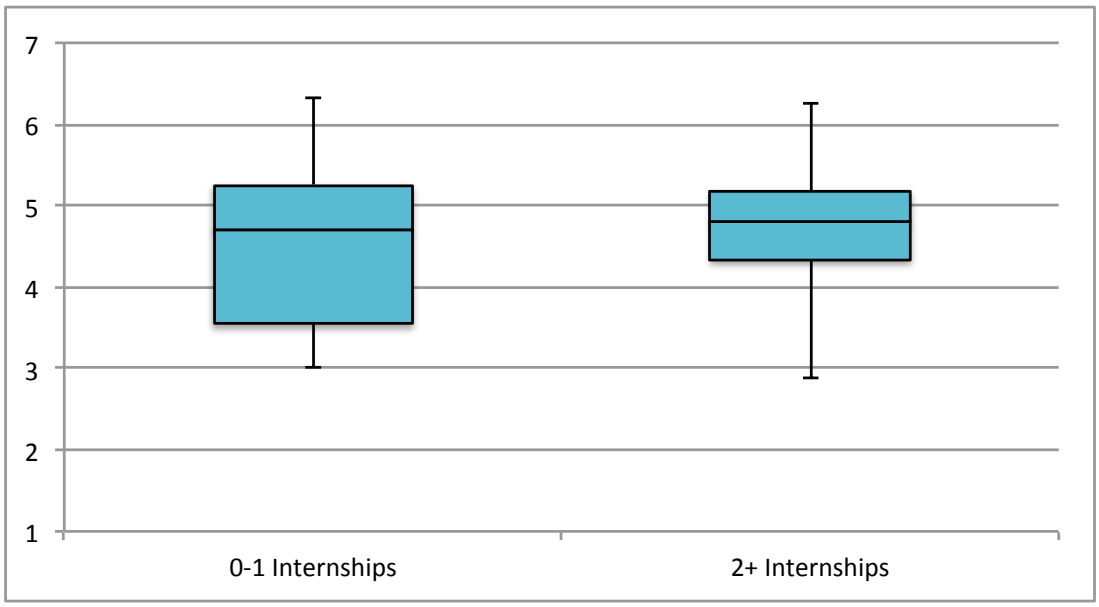

Figure 6.14: Box-plot of average helpfulness of text by experience.

clustered thoughts about the suggestion text. However, overall, regardless of industry experience, the average approval rating was relatively similar - and indicated room for improvement. Therefore, there is no real correlation between internship experience and how helpful the text is for a given suggestion.

6.3.3 Helpfulness by suggestion type

As with usefulness, the suggestion type was the best indicator of how experts felt about the suggestion text. Suggestion Type 2 had the best score as seen in Table 6.5. I believe this is once again due to the simplicity of the suggestion. It is much simpler

to imagine the impact of Suggestion Type 2 prior to having seen the refactored code when the section of code is referencing the same variable or variables.

Table 6.5: Average helpfulness of text by suggestion type

Suggestion Type	Score (out of 7)	Score (as percentage)
Type 1	4.59	59.9%
Type 2	5.11	68.5%
Type 3	4.56	59.3%

In this category, Suggestion Type 1 and Suggestion Type 3 fared equally poorly. What was particularly interesting was the spread of the average scores of Suggestion Type 1 from 4.2 to 4.9 (or 54.3% to 65.6%). The 10% difference in score percentage was surprising given that the reason for all 16 messages was nearly identical. Although each suggestion had unique parameters and return values, the reason the suggestion was identified was the same for all suggestions of Suggestion Type 1.

I believe the reason the helpfulness of the text had such a large range was due to a skew based on perceived usefulness of the suggestion. This is supported by the scatter plot in Figure 6.15 which plots the suggestion's overall usefulness on the x-axis against the helpfulness of the text on the y-axis. Each point on the plot represents the average scores for usefulness and helpfulness for one suggestion. The upward trajectory of the linear regression that fits this scatter plot shows there is a relationship between perceived usefulness and helpfulness of a suggestion. Even with the skew, Suggestion Type 1 helpfulness averaged nearly half a point lower (or 7% lower).

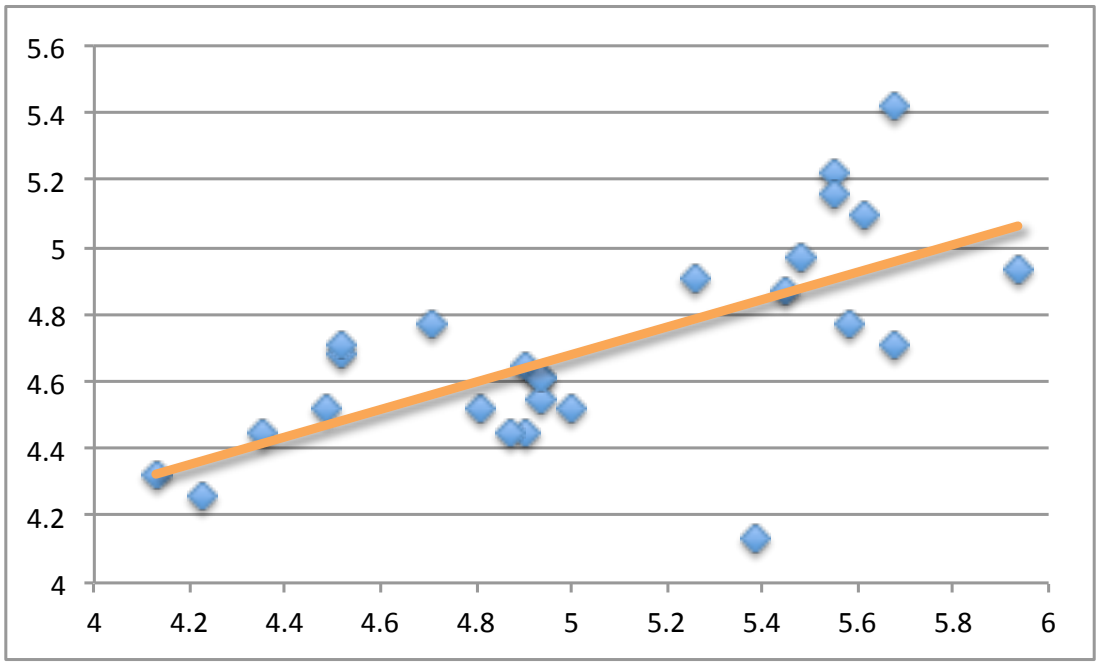

Figure 6.15: Scatter plot showing the suggestion's overall usefulness versus the helpfulness of the suggestion's text.

6.4 Overall Value

Overall, the experts rated decomposition as valuable in both education and industry as seen in Table 6.6. The table shows how important the experts believed decomposition is in education, early education, and industry. As a whole, experts scored the importance of decomposition as 6.52 (or 91.9%). Although early education received the lowest score of the three categories, it still averaged 6.39 out of 7 in importance.

Table 6.6: Importance of decomposition

Category	Score (out of 7)	Score (as percentage)
Education	6.52	91.9%
Early Education	6.39	89.7%
Industry	6.65	94.1%
Overall	6.52	91.9%

Students in CPE 101 valued decomposition slightly less, rating it a 6 out of 7 (or 83.3%). I believe this was in part due to lack of context - given that they have never worked with source code files over 200 lines. This claim is supported by the fact that students were much more likely to use the tool if they were graded on decomposition as seen in Table 6.7. This indicates that while students value code decomposition, at an introductory level they are only willing to put forth the effort to improve it if they are graded on it. If students will only decompose code when they get graded on decomposition, a tool like *Earthworm* plays a crucial role in being able to scale the effort of teaching decomposition to become a graded component of courses.

Table 6.7: Likeliness to use the tool if decomposition is graded vs not graded

	Likeliness to use (out of 7)
Decomposition Graded	5.78
Decomposition Not Graded	3.72

6.5 Free Response Feedback

This section discusses some of the feedback from the free response questions in both the student and expert surveys.

As a whole, the students were pleased with the improvements to the code after refactoring the suggestions. However, there was a learning curve that could have been eased with a more user friendly output. The main points of contention for those who filled out the student survey were the following.

- There were too many suggestions output to the screen at once.
- There was a need for more thorough descriptions of why each suggestion was being output.

The students provided creative solutions, such as creating a user manual, for solving these issues that have been detailed in Section 7.3.

The experts reiterated the students' sentiment regarding the text needing a better description. They felt the reasoning could have been unique to each suggestion to help understand why that suggestion was generated. Another common point of feedback was to decrease the repetitiveness of the suggestions. For example, suggestions 1 through 4 for Code Sample 2 were nearly identical. Experts felt those suggestions could have been condensed into one suggestion. Lastly, a handful of experts felt some of the suggestions over-decomposed the code.

Overall, the experts found the tool to be useful and noted that the tool provided suggestions that generated significantly more readable code. Although some found the individual suggestions excessive or too large, putting them together made the final code after refactoring more concise.

6.6 Summary

This chapter demonstrates there is a great deal of room for improvement, but also shows the current value and the future potential of *Earthworm*. It reiterates that coders at all levels consider decomposition to be an important skill. *Earthworm* can help enforce this in introductory computer science courses by easing the process of providing feedback on student's decomposition - instead of placing all of the burden on the professor. Given that students in introductory classes focus on decomposition only when it affects their grade, it is important to create that incentive while providing them the tools to learn how to improve their decomposition.

Chapter 7

FUTURE WORK

This chapter discusses the improvements that can be made to *Earthworm*. The opportunities for improvement relate to three facets of *Earthworm*: algorithms for generating suggestions, metrics to measure complexity and decomposition, and the presentation of suggestion to the user. The last section discusses opportunity for future integration with IDEs.

7.1 Improving Suggestions

There are a handful of avenues that I believe can be explored to improve the quality of the suggestions generated by *Earthworm*.

First, it would be beneficial to explore other approaches to generate suggestions. There are two approaches I thought of during development that I was not able to implement in this version of *Earthworm*. First, the approach discussed in Section 4.3 which examines multiple consecutive instructions with the same local references should be made to work on multiline statements. Another approach worth exploring is finding all the conditionals where every branch has the same set of defined variables.

There are a few additions to the criteria for eliminating suggestions discussed in Section 4.6 that could improve suggestions. First, the criteria for eliminating should be based on the number of lines in the function. This could allow for adaptability such as an adjustable scale of minimum and maximum number of function parameters and return values. Theoretically, this could be fine-tuned on a function-level basis using some metric of complexity. In addition, developing a method to condense suggestions would enable users to focus on the important suggestions. It would decrease clutter,

therefore improving the user experience. This would require an ability to determine the better of two similar suggestions.

Lastly, to improve the grouping algorithm in Section 4.5.1, it could be beneficial to create an option that checks every possible combination of suggestions for a given set of line numbers to find the optimal groups that would create high-quality suggestions. Instead of grouping the lines based on gaps, *Earthworm* could exhaust all combinations to find the optimal suggestions. As with the last feature, it would require an ability to determine the best suggestions.

All of these changes could lead to additional suggestions such as the conditional nested within a exception handling block that was previously not suggested for refactoring[1]. As mentioned in Section 5.4, if a user were to refactor this code block into another function, it could cause other suggestions to no longer appear if the tool were run again on the code sample. In other words, by adding more approaches to generate suggestions, it would require identifying high complexity suggestions getting rid of lower complexity suggestions.

7.2 Investigating Metrics

During development, one of the areas I felt could use the most exploration was finding metrics that might work better with the task at hand. Recall Section 4.7 identified the shortcomings of cyclomatic complexity's ability to identify as complex long functions composed of straight line code. Additionally, the other approach investigated in Section 4.7, using cyclomatic complexity with the relative line number, created a metric that increased with actual complexity, but that was highly correlated to the order of statements.

Investing time into exploring other metrics that could be universal to all pro-

[1]Reference line 8 through 26 in source code in Appendix A.1.1 prior to refactoring.

grams would provide two main benefits. First, it could improve the identification of suggestions. Instead of cyclomatic complexity, it could be used with slicing to try to generate suggestions. Additionally, it would also allow for the development of a metric scoring system that could streamline grading code decomposition.

7.3 Improving the Output

The primary area in need of improvement, according to the surveys from Chapter 5, is *Earthworm*'s output. Based on the student survey feedback there are multiple approaches to try to improve the output.

One change that would increase student comprehension about decomposition from using *Earthworm* is outputting a unique reason for each suggestion based on the properties of the code highlighted by the suggestion. For example, in Section 5.1 we were shown straight line code that decreases in complexity due to prior control flow that sets the value for `inFile`. The current message states "Removing these instructions decreases number of paths of execution in the function - making the code more readable and testable." A more refined message might state "Moving this straight-line code into a function separates it from the prior control flow that sets the value of `inFile` - making the code more readable and testable."

Additionally, students felt there were too many suggestions output to the screen. *Earthworm* could output only one or two suggestions to the screen at a time. The rest could be output to a file so students have more flexibility to choose to revisit those suggestions later.

The text file could also include the code that needs to be pulled out into another function. The issue with line numbers is that they change after making a single refactor. If students have to rerun *Earthworm* after every change, it slows their development cycle, decreasing the incentive to use the tool. Therefore, outputting

the lines of code in the suggestion to a text file reduces the student's overhead. Two alternative solutions are reversing the order the suggestions are output to encourage making changes from the bottom of the source file or having *Earthworm* augment the source file with comments surrounding the lines for each suggestion.

Some students suggested making a manual that outlines directions on how to use the tool while debugging. While improving the reasoning for specific suggestions should certainly help them understand why that specific suggestion was generated, making a manual could allow for definitions of basic terminology and create a space for examples and instructions on how to use the tool.

7.4 IDE Integration

An eventual goal for *Earthworm* is to allow integration with an IDE such as PyCharm. The tool could be part of the `Refactor` toolkit provided. The suggestions could be displayed with a more user friendly interface by embedding them into the IDE. Additionally, the suggestions would adjust as the user refactors the program instead of requiring the tool to be rerun between changes. This would also allow users to use the IDE's `Refactor` toolkit to perform the final step of transforming the code.

Chapter 8

CONCLUSION

Code decomposition is integral to programming yet often overlooked in introductory college courses due to a lack of resources. This paper presents *Earthworm*, a tool for students that generates suggestions on how to improve the code decomposition of their Python code.

Given a program as input, *Earthworm* presents the user with suggestions that include the lines of code that can be refactored into a new function along with the new function's arguments and return values.

Overall, as seen in the analysis, there is strong evidence to believe there is a correlation between using *Earthworm* and students building confidence in decomposing code. The suggestions *Earthworm* output are generally useful; they lead to functions that are easier to test, resulting in code that exhibits better decomposition.

Although there is opportunity to improve the suggestions output by *Earthworm* and improve its user interface, it lays a foundation in proving the value of such a tool. It can reduce the overhead of teaching code decomposition in a way that does not require extensive background knowledge from students and limits the overhead from instructors to allow it to be integrated more seamlessly into existing introductory computer science courses.

BIBLIOGRAPHY

[1] Cal Poly Github. http://www.github.com/CalPoly.

[2] Earthworm github. https://github.com/gargn/thesis.

[3] Eclipse documentation. https://help.eclipse.org, May 2017.

[4] Pycharm documentation. https://www.jetbrains.com/help/pycharm/, May 2017.

[5] Pyflakes documentation. https://pypi.python.org/pypi/pyflakes, May 2017.

[6] Pylint documentation. https://www.pylint.org/, May 2017.

[7] Software developers: Occupational outlook handbook. May 2016.

[8] Surging demand and new approaches transform computer science education. https://www.acm.org/media-center/2017/february/sigcse-2017, June 2017.

[9] Visual studio documentation. https://msdn.microsoft.com/en-us/library/mt125495.aspx, May 2017.

[10] D. W. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[11] K. D. Cooper and L. Torczon. *Engineering A Compiler*. Elsevier, Inc., 2nd edition, 2012.

[12] L. M. Fisher. Booming enrollments. *Commun. ACM*, 59(7):17–18, Jun 2016.

[13] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on*

Principles of Programming Languages, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[14] A. Lakhotia and J. C. Deprez. Restructuring functions with low cohesion. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 36–46, Oct 1999.

[15] F. Lanubile and G. Visaggio. Extracting reusable functions by program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, April 1997.

[16] J. R. Lyle and K. B. Gallagher. A program decomposition scheme with applications to software modification and testing. In *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*, volume 2, pages 479–485 vol.2, Jan 1989.

[17] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[18] F. Tip. A survey of program slicing techniques. 1994.

[19] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, July 1981.

APPENDICES

Appendix A

CODE SAMPLES

A.1 Code Samples for Validation

The code samples listed below are referenced throughout the paper in various sections. These specific samples were also used during validation.

A.1.1 Code Sample #1

Listing A.1 contains all of the code for a program that blurs an image formatted as a PPM. The starting code contains all 100+ lines of code in one function, `main`. The suggestions generated by *Earthworm* are provided in Listing A.2. The refactored code after using *Earthworm* is provided in Listing A.3.

Listing A.1: Code Sample #1 for validation survey

```
1  from sys import *
2  import math
3
4  def main():
5      file_name = "error"
6      outFile = open("blurred.ppm", "w")
7
8      # Open file.
9      try:
10         if len(argv) == 2:
11             file_name = argv[1]
12             reach = 4
13
14             inFile = open(file_name, 'r')
15         elif len(argv) == 3:
16             file_name = argv[1]
17             reach = int(argv[2])
18
```

```
19              inFile = open(file_name, 'r')
20          else:
21              print("Usage: python blur.py <image> <OPTIONAL:reach>")
22              exit()
23      except IOError:
24          print("Unable to open %s" %file_name)
25          exit()
26
27      # Print header.
28      header = inFile.readline()
29      w_and_h = inFile.readline().split()
30      width = int(w_and_h[0])
31      height = int(w_and_h[1])
32      MAX_COMP_NUM = int(inFile.readline())
33      outFile.write(header + str(width) + " " + str(height) + "\n" + str(
            MAX_COMP_NUM) + "\n")
34
35      # Read file.
36      pixels = []
37      rgb = []
38      for line in inFile:
39          line = line.split()
40          for comp in line:
41              if len(rgb) != 3:
42                  rgb.append(comp)
43              if len(rgb) == 3:
44                  pixels.append(rgb)
45                  rgb = []
46
47      picture = []
48      i = 0
49      for row in range(height):
50          row = []
51          for col in range(width):
52              row.append(pixels[i])
53              i += 1
54          picture.append(row)
55
56      cCol = 0
57      cRow = 0
58      for pixel in pixels:
59          totalR = 0
60          totalB = 0
61          totalG = 0
62          totalP = 0
63
64          # Calculate lower bounds of neighborhood.
65          lowCol = 0
66          if cCol - reach > 0:
```

```
67              lowCol = cCol - reach
68          lowRow = 0
69          if cRow - reach > 0:
70              lowRow = cRow - reach
71
72          # Calculate upper bounds of neighborhood.
73          upCol = width
74          if cCol + reach < upCol:
75              upCol = cCol + reach
76          upRow = height
77          if cRow + reach < upRow:
78              upRow = cRow + reach
79
80          # Calculate neighborhood totals.
81          rows = picture[lowRow:upRow]
82          for row in rows:
83              cols = row[lowCol:upCol]
84
85              for neighbor in cols:
86                  totalR += int(neighbor[0])
87                  totalG += int(neighbor[1])
88                  totalB += int(neighbor[2])
89                  totalP += 1
90
91          red = int(totalR / totalP)
92          green = int(totalG / totalP)
93          blue = int(totalB / totalP)
94
95          # Write out new pixel.
96          newPixel = str(red) + " " + str(green) + " " + str(blue) + "\n"
97          outFile.write(newPixel)
98
99          # Updates position.
100         if cCol == width - 1:
101             cCol = 0
102             cRow += 1
103         else:
104             cCol += 1
105
106     inFile.close()
107     outFile.close()
108
109
110 if __name__ == '__main__':
111     main()
```

Listing A.2: Suggestions generated for Code Sample #1 by *Earthworm*

```
1  line 28-33 (main):
2          parameters: inFile, outFile
3          returns: height, width
4          reason: Removing these instructions decreases number of paths of
               execution in this function - making the code more readable
              and testable.
5
6  line 36-104 (main):
7          parameters: height, inFile, outFile, reach, width
8          reason: Multiple variables defined prior to these instructions
              are not used in these line numbers.
9
10 line 38-104 (main):
11         parameters: height, inFile, outFile, pixels, reach, rgb, width
12         reason: Multiple variables defined prior to these instructions
              are not used in these line numbers.
13
14 line 38-45 (main):
15         parameters: inFile, pixels, rgb
16         returns: pixels
17         reason: Removing these instructions decreases number of paths of
               execution in this function - making the code more readable
              and testable.
18
19 line 41-45 (main):
20         parameters: comp, pixels, rgb
21         returns: pixels
22         reason: Removing these instructions decreases number of paths of
               execution in this function - making the code more readable
              and testable.
23
24 line 49-54 (main):
25         parameters: height, i, picture, pixels, width
26         returns: picture
27         reason: Removing these instructions decreases number of paths of
               execution in this function - making the code more readable
              and testable.
28
29 line 58-104 (main):
30         parameters: cCol, cRow, height, outFile, picture, pixels, reach,
               width
31         reason: Removing these instructions decreases number of paths of
               execution in this function - making the code more readable
              and testable.
32
33 line 81-97 (main):
34         parameters: lowCol, lowRow, outFile, picture, totalB, totalG,
              totalP, totalR, upCol, upRow
```

```
35          reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
36
37  line 82-97 (main):
38          parameters: lowCol, outFile, rows, totalB, totalG, totalP,
                totalR, upCol
39          reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
40
41  line 91-97 (main):
42          parameters: outFile, totalB, totalG, totalP, totalR
43          reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
44
45  line 100-104 (main):
46          parameters: cCol, cRow, width
47          returns: cCol, cRow
48          reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
```

Listing A.3: Code Sample #1 for validation survey after using *Earthworm*

```
1  from sys import *
2  import math
3
4
5  # Prints header.
6  def print_header(inFile, outFile):
7      header = inFile.readline()
8      w_and_h = inFile.readline().split()
9      width = int(w_and_h[0])
10     height = int(w_and_h[1])
11     MAX_COMP_NUM = int(inFile.readline())
12     outFile.write(header + str(width) + " " + str(height) + "\n" + str(
           MAX_COMP_NUM) + "\n")
13     return height, width
14
15
16 # Read file.
17 def read_file(inFile, pixels, rgb):
18     pixels = []
19     rgb = []
20     for line in inFile:
21         line = line.split()
22         for comp in line:
```

```
23              if len(rgb) != 3:
24                  rgb.append(comp)
25              if len(rgb) == 3:
26                  pixels.append(rgb)
27                  rgb = []
28      return pixels
29
30
31  # Format pixels as a list of list as laid out in the picture.
32  def get_formatted_pixels(height, pixels, width):
33      picture = []
34      i = 0
35      for row in range(height):
36          row = []
37          for col in range(width):
38              row.append(pixels[i])
39              i += 1
40          picture.append(row)
41      return picture
42
43
44  # Calculate red, green, blue values.
45  def calculate_rgb(outFile, totalB, totalG, totalP, totalR):
46      red = int(totalR / totalP)
47      green = int(totalG / totalP)
48      blue = int(totalB / totalP)
49
50      # Write out new pixel.
51      newPixel = str(red) + " " + str(green) + " " + str(blue) + "\n"
52      outFile.write(newPixel)
53
54
55  # Calculate neighborhood totals.
56  def calculate_neighbors(lowCol, lowRow, outFile, picture,
57                          totalB, totalG, totalP, totalR, upCol, upRow):
58      rows = picture[lowRow:upRow]
59      for row in rows:
60          cols = row[lowCol:upCol]
61
62          for neighbor in cols:
63              totalR += int(neighbor[0])
64              totalG += int(neighbor[1])
65              totalB += int(neighbor[2])
66              totalP += 1
67      calculate_rgb(outFile, totalB, totalG, totalP, totalR)
68
69
70  # Updates position.
71  def update_position(cCol, cRow, width):
```

```
72      if cCol == width - 1:
73          cCol = 0
74          cRow += 1
75      else:
76          cCol += 1
77
78
79  # Processes file.
80  def process_file(inFile, outFile, reach, width):
81      pixels = read_file(inFile, pixels, rgb)
82      picture = get_formatted_pixels(height, pixels, width)
83
84      cCol = 0
85      cRow = 0
86      for pixel in pixels:
87          totalR = 0
88          totalB = 0
89          totalG = 0
90          totalP = 0
91
92          # Calculate lower bounds of neighborhood.
93          lowCol = 0
94          if cCol - reach > 0:
95              lowCol = cCol - reach
96          lowRow = 0
97          if cRow - reach > 0:
98              lowRow = cRow - reach
99
100         # Calculate upper bounds of neighborhood.
101         upCol = width
102         if cCol + reach < upCol:
103             upCol = cCol + reach
104         upRow = height
105         if cRow + reach < upRow:
106             upRow = cRow + reach
107
108         calculate_neighbors(lowCol, lowRow, outFile, picture,
109             totalB, totalG, totalP, totalR, upCol, upRow)
110
111         cCol, cRow = update_position(cCol, cRow, width)
112
113
114 def main():
115     file_name = "error"
116     outFile = open("blurred.ppm", "w")
117
118     # Open file.
119     try:
120         if len(argv) == 2:
```

```
121             file_name = argv[1]
122             reach = 4
123 
124             inFile = open(file_name, 'r')
125         elif len(argv) == 3:
126             file_name = argv[1]
127             reach = int(argv[2])
128 
129             inFile = open(file_name, 'r')
130         else:
131             print("Usage: python blur.py <image> <OPTIONAL:reach>")
132             exit()
133     except IOError:
134         print("Unable to open %s" %file_name)
135         exit()
136 
137     # Processes file.
138     height, width = print_header(inFile, outFile)
139     process_file(inFile, outFile, reach, width)
140 
141     inFile.close()
142     outFile.close()
143 
144 
145 if __name__ == '__main__':
146     main()
```

A.1.2 Code Sample #2

Listing A.4 contains part of the solution for a command line version of 2048. The solution logic is adapted from github repository `yangshan/2048-python`. The suggestions generated by *Earthworm* are provided in Listing A.5. The refactored code after using *Earthworm* is provided in Listing A.6.

Listing A.4: Code Sample #2 for validation survey

```
107 # Moves the board.
108 def move_board(game, direction):
109     has_shift = False
110     has_merge = False
111 
112     if direction == 'w':
113         print('... UP ...')
114         game = transpose(game)
```

```
115         game, has_shift = shift_left(game)
116         game, has_merge = merge(game)
117         game, _ = shift_left(game)
118         game = transpose(game)
119     elif direction == 'a':
120         print('... LEFT ...')
121         game, has_shift = shift_left(game)
122         game, has_merge = merge(game)
123         game, _ = shift_left(game)
124     elif direction == 's':
125         print('... RIGHT ...')
126         game = reverse(game)
127         game, has_shift = shift_left(game)
128         game, has_merge = merge(game)
129         game, _ = shift_left(game)
130         game = reverse(game)
131     elif direction == 'z':
132         print('... DOWN ...')
133         game = reverse(transpose(game))
134         game, has_shift = shift_left(game)
135         game, has_merge = merge(game)
136         game, _ = shift_left(game)
137         game = transpose(reverse(game))
138     else:
139         print('... INVALID MOVE ...')
140 
141     made_move = has_shift or has_merge
142     return (game, made_move)
143 
144 
145 # Prints the board.
146 def print_board(board):
147     for row in board:
148         for col in row:
149             print('{0} '.format(col), end=' ')
150         print()
151     print()
152 
153 
154 def main():
155     # Initialize board.
156     board = new_game(size=4)
157     add_two(board)
158     add_two(board)
159     print_board(board)
160 
161     # Loop until game state ended.
162     game_state = 0
163 
```

```
164     while game_state == 0:
165         direction = input('Enter direction to move (w = UP, a = LEFT, s
                = RIGHT, z = DOWN): ')
166         board, made_move = move_board(board, direction)
167         if made_move:
168             add_two(board)
169 
170             # Prints the board.
171             print_board(board)
172 
173             # Set variables for next loop.
174             game_state = get_game_state(board)
175 
176     # Prints the results of the game.
177     if game_state == 1:
178         print('YOU WON!')
179     else:
180         print('YOU LOST')
181 
182 if __name__ == '__main__':
183     main()
```

Listing A.5: Suggestions generated for Code Sample #2 by *Earthworm*

```
1  line 114-118 (move_board):
2          parameters: game
3          returns: game, has_merge, has_shift
4          reason: The same set of variables are referenced in all
               instructions in the given line numbers.
5 
6  line 121-123 (move_board):
7          parameters: game
8          returns: game, has_merge, has_shift
9          reason: The same set of variables are referenced in all
               instructions in the given line numbers.
10 
11 line 126-130 (move_board):
12         parameters: game
13         returns: game, has_merge, has_shift
14         reason: The same set of variables are referenced in all
               instructions in the given line numbers.
15 
16 line 133-137 (move_board):
17         parameters: game
18         returns: game, has_merge, has_shift
19         reason: The same set of variables are referenced in all
               instructions in the given line numbers.
20 
21 line 157-159 (main):
```

```
22          parameters: board
23          reason: The same set of variables are referenced in all
                instructions in the given line numbers.
24
25  line 168-174 (main):
26          parameters: board
27          returns: game_state
28          reason: The same set of variables are referenced in all
                instructions in the given line numbers.
29
30  line 177-180 (main):
31          parameters: game_state
32          reason: Removing these instructions decreases number of paths of
                 execution in this function - making the code more readable
                and testable.
```

Listing A.6: Code Sample #2 for validation survey after using *Earthworm*

```
107 # Moves the board up.
108 def move_board_up(game):
109     game = transpose(game)
110     game, has_shift = shift_left(game)
111     game, has_merge = merge(game)
112     game, _ = shift_left(game)
113     game = transpose(game)
114
115
116 # Moves the board left.
117 def move_board_left(game):
118     game, has_shift = shift_left(game)
119     game, has_merge = merge(game)
120     game, _ = shift_left(game)
121
122
123 # Moves the board right.
124 def move_board_right(game):
125     game = reverse(game)
126     game, has_shift = shift_left(game)
127     game, has_merge = merge(game)
128     game, _ = shift_left(game)
129     game = reverse(game)
130
131
132 # Moves the board down.
133 def move_board_down(game):
134     game = reverse(transpose(game))
135     game, has_shift = shift_left(game)
136     game, has_merge = merge(game)
137     game, _ = shift_left(game)
```

```
138     game = transpose(reverse(game))
139
140
141 # Moves the board.
142 def move_board(game, direction):
143     has_shift = False
144     has_merge = False
145
146     if direction == 'w':
147         print('... UP ...')
148         move_board_up()
149     elif direction == 'a':
150         print('... LEFT ...')
151         move_board_left()
152     elif direction == 's':
153         print('... RIGHT ...')
154         move_board_right()
155     elif direction == 'z':
156         print('... DOWN ...')
157         move_board_down()
158     else:
159         print('... INVALID MOVE ...')
160
161     made_move = has_shift or has_merge
162     return (game, made_move)
163
164
165 # Prints the board.
166 def print_board(board):
167     for row in board:
168         for col in row:
169             print('{0} '.format(col), end=' ')
170         print()
171     print()
172
173
174 # Initializes board.
175 def init_board(board):
176     add_two(board)
177     add_two(board)
178     print_board(board)
179
180
181 # Makes move on the board.
182 def make_move(board):
183     add_two(board)
184
185     # Prints the board.
186     print_board(board)
```

```
187
188     # Set variables for next loop.
189     return get_game_state(board)
190
191
192 # Prints result of the game.
193 def print_result(game_state):
194     if game_state == 1:
195         print('YOU WON!')
196     else:
197         print('YOU LOST')
198
199
200 def main():
201     # Initialize board.
202     board = new_game(size=4)
203     init_board(board)
204
205     # Loop until game state ended.
206     game_state = 0
207
208     while game_state == 0:
209         direction = input('Enter direction to move (w = UP, a = LEFT, s
                = RIGHT, z = DOWN): ')
210         board, made_move = move_board(board, direction)
211         if made_move:
212             game_state = make_move(board)
213     print_result(game_state)
214
215 if __name__ == '__main__':
216     main()
```

A.1.3 Code Sample #3

Listing A.7 is a crossword puzzle solver. The two functions provided search for words in the rows (forwards or backwards) and for words in the columns (up or down). The suggestions generated by *Earthworm* are provided in Listing A.8. The refactored code after using *Earthworm* is provided in Listing A.9.

Listing A.7: Code Sample #3 for validation survey

```
1  def make_puzzle(Puzzle):
2      puzzle = []
3      for i in range(0,10):
4          puzzle.append(Puzzle[i*10:i*10+10])
5      return puzzle
6
7  def make_words(Words):
8      return Words.split()
9
10
11 def check_rows(puzzle, word):
12     newword = []
13     j = len(word)
14     while j > 0:
15         newword.append(word[j-1])
16         j -= 1
17     backWord = ''.join(newword)
18     for row in range(len(puzzle)):
19         if word in puzzle[row]:
20             length = 0
21             col = 0
22             for char in puzzle[row]:
23                 letter = word[length]
24                 if char != letter:
25                     length = 0
26                     letter = word[length]
27                 if char == letter:
28                         length += 1
29                         if len(word) == length:
30                             col -= (length-1)
31                             place = ['(FORWARD)', row, col]
32                             return place
33                 col += 1
34         if backWord in puzzle[row]:
35             length = 0
36             col = 0
37             for char in puzzle[row]:
38                 letter = backWord[length]
39                 if char != letter:
40                     length = 0
41                     letter = backWord[length]
42                 if char == letter:
43                     length += 1
44                     if len(backWord) == length:
45                         place = ['(BACKWARD)', row, col]
46                         return place
47                     col   += 1
48
```

```
49 def check_cols(puzzle, word):
50     newword = []
51     j = len(word)
52     while j > 0:
53         newword.append(word[j-1])
54         j -= 1
55     backWord = ''.join(newword)
56     newpuzzle = []
57     for col in range(len(puzzle)):
58         new = []
59         for row in range(len(puzzle)):
60             new.append(puzzle[row][col])
61         newpuzz = ''.join(new)
62         newpuzzle.append(newpuzz)
63     for col in range(len(newpuzzle)):
64         if word in newpuzzle[col]:
65             length = 0
66             row = 0
67             for char in newpuzzle[col]:
68                 letter = word[length]
69                 if char != letter:
70                     length = 0
71                     letter = word[length]
72                 if char == letter:
73                     length += 1
74                     if len(word) == length:
75                         row -= (length-1)
76                         place = ['(DOWN)', row, col]
77                         return place
78                 row += 1
79         if backWord in newpuzzle[col]:
80             length = 0
81             row = 0
82             for char in newpuzzle[col]:
83                 letter = backWord[length]
84                 if char != letter:
85                     length = 0
86                     letter = backWord[length]
87                 if char == letter:
88                     length += 1
89                     if len(backWord) == length:
90                         place = ['(UP)', row, col]
91                         return place
92                 row += 1
```

Listing A.8: Suggestions generated for Code Sample #3 by *Earthworm* using the slow flag

```
1  line 23-26 (check_rows):
2          parameters: char, length, word
3          returns: length, letter
4          reason: Multiple variables defined prior to these instructions
               are not used in these line numbers.
5
6  line 27-32 (check_rows):
7          parameters: char, col, length, letter, row, word
8          returns: col, length, place
9          reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
10
11 line 34-47 (check_rows):
12         parameters: backWord, puzzle, row
13         returns: letter, place
14         reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
15
16 line 38-41 (check_rows):
17         parameters: backWord, char, length
18         returns: length, letter
19         reason: Multiple variables defined prior to these instructions
               are not used in these line numbers.
20
21 line 42-47 (check_rows):
22         parameters: backWord, char, col, length, letter, row
23         returns: length, place
24         reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
25
26 line 63-92 (check_cols):
27         parameters: backWord, newpuzzle, word
28         returns: place
29         reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
30
31 line 68-71 (check_cols):
32         parameters: char, length, word
33         returns: length, letter
34         reason: Multiple variables defined prior to these instructions
               are not used in these line numbers.
35
36 line 72-77 (check_cols):
```

```
37          parameters: char, col, length, letter, row, word
38          returns: length, place, row
39          reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
40
41  line 79-92 (check_cols):
42          parameters: backWord, col, newpuzzle
43          returns: place
44          reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
45
46  line 83-86 (check_cols):
47          parameters: backWord, char, length
48          returns: length, letter
49          reason: Multiple variables defined prior to these instructions
               are not used in these line numbers.
50
51  line 87-92 (check_cols):
52          parameters: backWord, char, col, length, letter, row
53          returns: length, place
54          reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
55
56  line 87-91 (check_cols):
57          parameters: backWord, char, col, length, letter, row
58          returns: length, place
59          reason: Removing these instructions decreases number of paths of
                execution in this function - making the code more readable
               and testable.
```

Listing A.9: Code Sample #3 for validation survey after using *Earthworm*

```
1  def make_puzzle(Puzzle):
2      puzzle = []
3      for i in range(0,10):
4          puzzle.append(Puzzle[i*10:i*10+10])
5      return puzzle
6
7  def make_words(Words):
8      return Words.split()
9
10 # Checks if the given letter is equal.
11 def check_letter(char, length, word):
12     letter = word[length]
13     if char != letter:
14         length = 0
```

```
15          letter = word[length]
16      return (length, letter)
17
18  # Checks if the word is found going forward.
19  def check_forward(char, col, length, letter, row, word):
20      place = None
21      if char == letter:
22          length += 1
23          if len(word) == length:
24              col -= (length-1)
25              place = ['(FORWARD)', row, col]
26      return (col, length, place)
27
28  # Checks if the word is found going backward.
29  def check_backword(backWord, char, col, length, letter, row):
30      place = None
31      if char == letter:
32          length += 1
33          if len(backWord) == length:
34              place = ['(BACKWARD)', row, col]
35      return (length, place)
36
37  # Tries to find the word going backwards.
38  def find_backward(backWord, puzzle, row):
39      if backWord in puzzle[row]:
40          length = 0
41          col = 0
42          for char in puzzle[row]:
43              letter = backWord[length]
44              if char != letter:
45                  length = 0
46                  letter = backWord[length]
47              length, place = check_backword(backWord, char, col, length,
                    letter, row)
48              if place:
49                  return place
50              col  += 1
51      return None
52
53  # Checks the rows of the puzzle.
54  def check_rows(puzzle, word):
55      newword = []
56      letter_word = len(word)
57      while letter_word > 0:
58          newword.append(word[letter_word-1])
59          letter_word -= 1
60      backWord = ''.join(newword)
61      for row in range(len(puzzle)):
62          if word in puzzle[row]:
```

```
63            length = 0
64            col = 0
65            for char in puzzle[row]:
66               length, letter = check_letter(char, length, word)
67               col, length, place = check_forward(char, col, length, letter
                     , row, word)
68               if place:
69                  return place
70               col += 1
71         place = find_backward(backWord, puzzle, row)
72         if place:
73            return place
74
75 # Checks if the word is found going down.
76 def check_down(char, col, length, letter, row, word):
77      place = None
78      if char == letter:
79         length += 1
80         if len(word) == length:
81            row -= (length-1)
82            place = ['(DOWN)', row, col]
83            return place
84      return (length, place, row)
85
86 # Checks if the word is found going up.
87 def check_up(backWord, char, col, length, letter, row):
88      place = None
89      if char == letter:
90         length += 1
91         if len(backWord) == length:
92            place = ['(UP)', row, col]
93            return place
94      return (length, place)
95
96 # Tries to find the word going up.
97 def find_up(backWord, col, newpuzzle):
98      if backWord in newpuzzle[col]:
99            length = 0
100           row = 0
101           for char in newpuzzle[col]:
102              length, letter = check_letter(char, length, backWord)
103              length, place = check_up(backWord, char, col, length, letter
                     , row)
104              if place:
105                 return place
106              row += 1
107
108 # Finds a word either up or down.
109 def find_up_down(backWard, newpuzzle, word):
```

```
110     for col in range(len(newpuzzle)):
111         if word in newpuzzle[col]:
112             length = 0
113             row = 0
114             for char in newpuzzle[col]:
115                 letter = word[length]
116                 length, letter = check_letter(char, length, word)
117                 length, place, row = check_down(char, col, length, letter,
                        row, word)
118                 if place:
119                     return place
120                 row += 1
121         place = find_up(backWord, col, newpuzzle)
122         if place:
123             return place
124 
125 # Checks the columns of the puzzle.
126 def check_cols(puzzle, word):
127     newword = []
128     j = len(word)
129     while j > 0:
130         newword.append(word[j-1])
131         j -= 1
132     backWord = ''.join(newword)
133     newpuzzle = []
134     for col in range(len(puzzle)):
135         new = []
136         for row in range(len(puzzle)):
137             new.append(puzzle[row][col])
138         newpuzz = ''.join(new)
139         newpuzzle.append(newpuzz)
140     return find_up_down(backWard, newpuzzle, word)
```

A.1.4 Code Sample #4

Listing A.10 is the `playgame` function for a command line version of Minesweeper. The final code is not provided for this code sample.

Listing A.10: Code Sample #4 for validation survey

```
141 def playgame():
142     gridsize = 9
143     numberofmines = 10
144 
145     currgrid = [[' ' for i in range(gridsize)] for i in range(gridsize)]
146 
```

```
147     grid = []
148     flags = []
149     starttime = 0
150 
151     helpmessage = ("Type the column followed by the row (eg. a5). "
152                    "To put or remove a flag, add 'f' to the cell (eg.
                           a5f).")
153 
154     showgrid(currgrid)
155     print(helpmessage + " Type 'help' to show this message again.\n")
156 
157     while True:
158         minesleft = numberofmines - len(flags)
159         prompt = input('Enter the cell ({} mines left): '.format(
                minesleft))
160         result = parseinput(prompt, gridsize, helpmessage + '\n')
161 
162         message = result['message']
163         cell = result['cell']
164 
165         if cell:
166             print('\n\n')
167             rowno, colno = cell
168             currcell = currgrid[rowno][colno]
169             flag = result['flag']
170 
171             if not grid:
172                 grid, mines = setupgrid(gridsize, cell, numberofmines)
173             if not starttime:
174                 starttime = time.time()
175 
176             if flag:
177                 # Add a flag if the cell is empty
178                 if currcell == ' ':
179                     currgrid[rowno][colno] = 'F'
180                     flags.append(cell)
181                 # Remove the flag if there is one
182                 elif currcell == 'F':
183                     currgrid[rowno][colno] = ' '
184                     flags.remove(cell)
185                 else:
186                     message = 'Cannot put a flag there'
187 
188             # If there is a flag there, show a message
189             elif cell in flags:
190                 message = 'There is a flag there'
191 
192             elif grid[rowno][colno] == 'X':
193                 print('Game Over\n')
```

```
194                     showgrid(grid)
195                     if playagain():
196                         playgame()
197                     return
198
199                 elif currcell == ' ':
200                     showcells(grid, currgrid, rowno, colno)
201
202                 else:
203                     message = "That cell is already shown"
204
205                 if set(flags) == set(mines):
206                     minutes, seconds = divmod(int(time.time() - starttime),
                        60)
207                     print(
208                         'You Win. '
209                         'It took you {} minutes and {} seconds.\n'.format(
                            minutes,
210                                 seconds))
211                     showgrid(grid)
212                     if playagain():
213                         playgame()
214                     return
215
216             showgrid(currgrid)
217             print(message)
218
219 def main():
220     playgame()
```