IMPROVING INTRODUCTORY COMPUTER SCIENCE EDUCATION WITH DRACO

A Thesis

Presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements of the Degree

Master of Science in Computer Science

by

Mike Dongyub Ryu

June 2018

COMMITTEE MEMBERSHIP

TITLE:	Improving Introductory Computer Science Education with DRaCO
AUTHOR:	Mike Dongyub Ryu
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	John B. Clements, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	David S. Janzen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Franz J. Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

Improving Introductory Computer Science Education with DRaCO

Mike Dongyub Ryu

Today, many introductory computer science courses rely heavily on a specific programming language to convey fundamental programming concepts. For beginning students, the cognitive capacity required to operate with the syntactic forms of this language may overwhelm their ability to formulate a solution to a program.

We recognize that the introductory computer science courses can be more effective if they convey fundamental concepts without requiring the students to focus on the syntax of a programming language. To achieve this, we propose a new teaching method based on the Design Recipe and Code Outlining (DRaCO) processes. Our new pedagogy capitalizes on the algorithmic intuitions of novice students and provides a tool for students to externalize their intuitions using techniques they are already familiar with, rather than with the syntax of a specific programming language. We validate the effectiveness of our new pedagogy by integrating it into an existing CS1 course at California Polytechnic State University, San Luis Obispo. We find that the our newly proposed pedagogy shows strong potential to improve students' ability to program.

ACKNOWLEDGMENTS

I give my utmost thanks to:

- My wife and best friend, Jessica Ryu, for all of her love, patience, support, and sacrifices. I dedicate my work here to you, my love!
- My parents, my sister Joy, and my family-in-law for their never-ending love and support.
- My advisor, Dr. John Clements, for his infinite patience and always telling me exactly what I needed to hear so I may succeed.
- Dr. David Janzen and Dr. Franz Kurfess, for being wonderful teachers and providing much help in and outside of classrooms.
- Professor Kurt Voelker, for his unmatched support and mentorship.
- Professor Paul Hatalsky and Christopher Siu for providing their amazing course material and infrastructure supporting all my work in CPE-101.
- All students of my Winter 2018 CPE-101 section who happily partook in my experiment and made this thesis possible.
- Tobias Bleisch, who is one of the most exemplary graduate students Cal Poly CSSE has ever seen, for all his support and help through my research.
- Tram Lai, Vivian Fong, and Jules Sulpico for being amazing friends who tolerated me.
- Evan Ovadia, for being a fantastic mentor for all of my years at Cal Poly.
- Last, but not least – Kyle Mulligan, for an amazing friendship, laughter, and all the burritos we shared together.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

In this chapter, we introduce our work with an illustration of the personal motivation that drives us to develop and present our work.

1.1 Motivation

Since its emergence as a unique discipline in early 1960s [1], computer science as a field of study never ceased to accelerate its growth. Today, the industry it has spawned is growing faster than all other industries [2]. It is undeniable that the contributions of computer science to the "widespread proliferation of emerging information and communication technologies" accelerated the coming of the Information Age in the early twenty-first century [3]. However, there has been limited advancement in the methods to teach such a discipline, even as other artifacts of computer science research have changed the world many times over within the past half-century.

1.1.1 An Anecdote on *Why*

To understand our motivation, it's important to examine how programmers today found their beginning, why that paradigm hasn't changed, and what *programming* really means in its most fundamental form.

1.1.1.1 How We All Started

Many established professionals in the field of computer science today probably remember their first few attempts at programming.

Regardless of whether it all started with a few lines of a C program that displayed out `Hello, World!` on the computer screen, or a few blocks snapped together in *Scratch* [4], many will agree that a prominent source of confusion originated from syntactic details of the first programming language they learned. The difference between brackets (`[]`) and parentheses (`()`) (in the case of the C program), or how *Events* blocks cannot be used within *Control* blocks (in the case of *Scratch* [5]), or the like, might seem obvious now. However, it's likely that many of today's professionals would admit to having had trouble understanding these syntax forms at first.

While the computer science as a field has grown dramatically in complexity over the years, we still observe this pattern of students struggling with the syntax of a programming language in many introductory classrooms. Why is it that the early computing education has not yet escaped the pattern of, as Stephen Bloch has put it, "drowning the students with syntax" [6]?

1.1.1.2 The Fundamental Definition of *Programming*

Within the context of early computer science education, the abilities that most instructors aim to pass on to their pupils consist not of any particular syntax of a programming language, but rather the analytic skills that are required to solve challenging problems using an algorithmic process called *programming*. What, then, is *programming*? Is it not, in its most fundamental form, a process of planning out stages of execution for the solution to a given problem? If that were true, why do so many programmers recall struggling with the syntax of a language in

introductory *programming* courses more prominently than they remember the challenges they faced in planning their solutions to algorithmic problems?

With this, we aim to highlight that the beginning computer science students have strong and useful intuitions on algorithmic thinking, but their intuitions are often crippled by traditional teaching methods that inundate students' cognitive capacities with syntax of an unfamiliar language. We illustrate this issue with an example provided in the following subsection.

1.1.1.3 The Party Guest List Problem

Imagine putting together a guest list for a big party. Perhaps the party we are throwing is very exclusive and we only want certain guests to attend, or we are simply generous hosts who would like to send thank-you notes later. In any case, we would want to make sure that each guest to our party only has one entry in the guest list, assuming no one we invite has the same name. This problem of checking for a duplicate entry in a list of names is simple enough to be asked of any student at a secondary level of education (middle or high school). Nevertheless, we have observed that when such a problem is asked in a programming course, the perceived difficulty of the problem seems to skyrocket.

For instance, consider the following example of a guest list shown below:

```
[Jess, Mike, Joy, Kyle, John, Toby, LaMarion, John, Joy]
```

If we ask middle school students how they may go about making sure we only have each guest's name appear only once in our guest list (again, assuming that all guests have unique names), they will not find it troublesome to draw up some sort of a plan on how to check for the name(s) that appear more than once. Perhaps

some of them will explain how they would look at each of the guest's name that appears in the list and look for every other guests' name to see if there is a match for a duplicate, which is a logically sound solution.

However, if we ask the same exact question to introductory programming students who have recently learned about list and strings, we will easily find many of students who struggle to come up with an answer. Perhaps the programming language in use is too low-level to effectively represent the problem, or the students' lack of proficiency with the language in use hinders their algorithmic thought process. This phenomenon is surprisingly easy to observe when assisting novice computer science students, even when the question does not require them to construct a working program to do so.

We are not mandating that all tasks in a programming course be approached initially with middle-school-level intuitions. In fact, it is reasonable to assume that many intuitive thoughts from a middle school student may be logically insufficient or unsound for many programming tasks students at higher level institutions face. Instead, we are attempting to simply illustrate the lack of reliance on planning out stages of execution in the beginning students. If the middle school students had some intuitions on this simple problem, students at college-level must also have some intuitions, likely to be more mature and logically sound, shaped by the concepts they learned in the course so far. Nevertheless, we frequently observe only a small subset of the beginning students thinks through the whole problem and plan what they are about to code before they begin typing away.

Certainly, it is not unnatural for students in a computer programming course to want to jump into practicing the skills they are learning, which is utilizing a syntax of a programming language to construct a working program. Nonetheless, what many students fail to recognize is that prematurely attempting to construct a snippet of code on a computer often hinders the process of planning out stages of execution of a logically sound solution to the given problem.

In terms of didactics in early computer science education, the importance of analytic skills and algorithmic thinking often dwarfs any significance in the act of learning and memorizing the syntax of some programming language. Nevertheless, we continue to observe alarming insufficiency in the efforts to harness the students' existing intuition to grow into maturely developed analytic skills and algorithmic thinking. The algorithmic problems presented to the students in introductory courses—such as the one illustrated above—are perfect opportunities for the students to apply their intuition on the 'process of planning out stages of execution for the solution to a given problem' (the fundamental definition of programming stated earlier). However, many educators fail to make this connection abundantly clear to their pupils as majority of the novice students still continue to perceive any problem given in a programming course as something that must be solved by typing code in some syntax of a programming language they are not yet familiar with.

1.1.1.4 Our Motivation

We want a framework in which the members of an introductory computer science classroom can effectively utilize their existing intuitions on programming to derive

and refine the core computer programming concepts. With such a framework, teaching and learning computer programming shall be feasible with minimal floundering caused by the syntax of any programming language.

1.1.2 Traditional Computer Science Education and Its Shortcomings

Here, we introduce Sally Fincher's 1999 publication that inspired our motivation. In it, Fincher briefly describes the origin of traditional computer science education as the acquisition of "the languages and techniques of programming for a specific purpose." She explains that most of those who initially learned how to program were "scientists, engineers, and mathematicians" [7]. For the most part, they did not care much for the complexities involved in programming as a discipline. They simply wanted to use computer programs as tools to solve their domain-specific problems, and it was sufficient for them to quickly pick up the syntax of a programming language and move on.

Contrastingly, Fincher identifies the modern computer science educators "no longer teach programming in order to get the computer to do something, but as a transferable skill in its own right" [7]. That is, when we teach programming in today's classrooms, we do emphasize the complexities in the discipline of programming more than the scientific, engineering, or mathematic problems we solve with it. She claims that the traditional teaching methods that follow how those scientists learned it—"via syntax, through the vehicle of a single language"—is limiting, because the "students get bogged down in the specifics of the chosen form" [7].

As Fincher demonstrates in her work, the limitations of traditional pedagogy in computer science were well recognized nearly three decades ago. However, despite some efforts to address the shortcomings and improve the effectiveness of the traditional methods in the past, mainstream introductory computer science courses at higher education institutions have not yet escaped the curricula that inevitably lead students to flounder, in varying degrees, in the syntax of a language students are not familiar with.

1.2 Outline of the Following Chapters

Here, we outline the subsequent chapters and provide brief previews of their content:

- In Chapter 2 (Related Work), we survey some of the existing alternatives that were proposed as improvements or replacements to the traditional methods and highlight how our contributions presented in this thesis differs.
- In Chapter 3 (Background), we present the strategies and concepts that already exist in computer science and non-computer science education we leverage in developing the new teaching method, as well as key definitions of the terms to be used in the following chapters.
- In Chapter 4 (Proposal of a New Teaching Method), we refine the goal and the premises of our new teaching method, then formally propose it to explain in detail our educational philosophy behind each component it contains within. This chapter is the main contribution of this thesis.

- In Chapter 5, (Implementation of the DRaCO-based Pedagogy), we provide details on how we implement the new teaching method to suit the existing introductory computer science course structure and curricula at California Polytechnic State University, San Luis Obispo. The implementation we illustrate in this chapter is used in validating the new teaching method we propose.
- In Chapter 6 (Validation of the DRaCO-based Pedagogy), we describe our execution of an A-B experiment to validate the new teaching methods and our implementation of it. We report whether our new teaching method demonstrates the positive educational impact we expect it to provide to the beginning computer science students, and whether it generates any negative emotional responses from the students.
- In Chapter 7 (Results and Discussion), we refine our thesis statements in order to apply rigorous statistical tests on the data gathered during our A-B experiment. Then, we conduct the analysis on the data to draw a conclusion regarding how effective our new teaching method is, and what negative consequences it may incur when being integrated it into an existing early computer science course.
- In Chapter 8 (Threats to Validity), we acknowledge certain weaknesses in our proposal of the new teaching method and disclose the details on potential criticisms or opportunities of reevaluation of our experiment and the data analysis.

- In Chapter 9 (Conclusion and Future Work), we conclude our research with a brief summary of the preceding chapters and provide directions for any further replications or refinements of our work.

Chapter 2

RELATED WORK

In this chapter, we survey some of previous efforts that have been extended to improve the early computer science to provide a context for the contribution of this thesis.

2.1 Pseudo-language-based Pedagogy

Despite the seemingly unshakable status quo of the past three decades of computer science education which demands “the vehicle of a single language” [7], many educators have chosen to explore the option of developing an alternative pedagogy that attempt to reduce the reliance on a single programming language.

We revisit Fincher’s 1999 publication and her presentation of a few notable implementations of the pseudo-language-based paradigm. Fincher presents these pedagogies as “syntax-free” [7] in her work. However, we recognize that they are not truly syntax-free, as they still utilize the syntax of pseudo-languages as a stepping stone for the syntax of formal programming languages.

First, she presents Richard Bornat and his methods detailed in his book *Programming from First Principles*. Based on the rationale “… it is the delusion that to learn a code is to learn to program which is truly harmful” [8], Bornat’s method is implemented with examples presented in ISWIM (short for “**I**f you **S**ee **W**hat **I** **M**ean;” an abstract programming language popularized through Landin’s seminal paper *The Next 700 Programming Languages* [9]) and exercises doable in paper and pencil, later to be used as notes when programming with a formal

programming language. The second method presented by Fincher is Russel Shackelford's use of pseudocode named RUSCAL as the "teaching vehicle" [7] to instill "algorithmic thinking" [10] in the mindsets of all students at Georgia Institute of Technology. Fincher classifies both of these implementations of "syntax-free" approaches as successful separation of *programming* from *coding* [7].

2.2 Initial Learning Environments

With the propagation of computer science education from the higher education institutions down to secondary and primary levels, popularity of Initial Learning Environments (ILEs) increased among introductory computer science educators throughout the past decade. Many of the ILEs answer the difficulties that arise from the syntax of a programming language with high interactivity and, in some cases, tools and environments that can help reduce syntactic mistakes.

In their 2009 paper, Fincher et al., enumerates the following three as the "leading" lLEs: *Alice*, *Greenfoot*, and *Scratch* [11]. Although these three ILEs are developed independently of each other using different technologies and target audiences, they all exhibit interactive graphical programs to engage students with high level of interactivity.

The idea of using interactive graphical elements in introductory programming courses has certainly existed for long. *Karel*—an educational programming language introduced in 1981 [12] designed for programming a robot to move around and perform simple in a two-dimensional GUI—exemplifies this. This paradigm of pairing a conventional practice of typing in the syntax of formal

programming language(s) with some graphical element is still employed in recently developed ILEs, such as in *Code Combat* [13].

However, a survey of more recent developments of ILEs reveals a different trend that started off with Scratch. Developed by MIT for “computer clubhouses” for younger students [11] [14], the programming pattern that Scratch offers stands out from the other ILEs. Instead of relying on a beginning programmer to type or drag-and-drop code into an editor in an error-prone way, it presents snippets of its proprietary syntax wrapped around in puzzle-piece-like blocks, such that the programmer can only put certain snippets together if the snippets belong together. This highly restrictive syntax greatly reduces chances for a novice programmer to make syntactic mistakes. Today, many ILEs utilize this puzzle-piece approach along with the interactive component to appeal to a younger audience. Some examples include Google’s *Blockly* [15], *Made with Code* [16], and Disney’s *Wayfinding with Moana* [17].

ILEs have proven their success in ‘initial learning.’ However, many of them lack proper segues to more generally-purposed programming environments that reach the ‘next level.’ For instance, MIT’s Scratch relies heavily on the graphical user interface and interactive software projects, while providing rather little opportunities for its user to organically move up to formal programming languages or environments that are not exclusively for initial learners. Here, we must acknowledge that there does exist rare exceptions like Google’s *Blockly* [15] that allow real-time translation from drag-and-drop ILEs to formal programming

languages, but such commitment to connect the ILEs to the next level programming environments is rather difficult to encounter.

Perhaps this is a fair drawback to expect, since the ILEs' focus is limited at initial learning and not much more. Nevertheless, this drawback may easily be a deal breaker for the higher education instructors, as many of them bear the responsibility of having to prepare their freshman pupils for the rest of their academic and professional programming career. Thus, there is simply no luxury to be able to reside only in the initial learning environments in higher education classrooms.

2.3 Multilingual Pedagogy

The Computer Science Division of University of California, Berkeley takes the opposite approach of the ones taken by syntax-free pedagogy or the ILEs. Instead of attempting to eliminate, minimize, or otherwise simplify the syntax of specific programming languages, they push a multilingual pedagogy that utilizes three different languages: Python, Scheme, and Structured Query Language (SQL).

They argue that their "goal is not to choose what language [students] use in [their] future studies and career," and that once the students have learned the essence of programming by observing the concepts employed by all three languages, they "will find that picking up a new programming language is but a few days' work" [18].

2.4 Planning-Based Pedagogy

We present a short-term and a long-term planning-based pedagogy here. First of the two is a single-lecture based methods by Castro and Fisler of Brown University. Motivated by the “tasks of developing and integrating programming plans” being “a recurring problem among programming students [19] [20] [21],” Castro and Fisler explore the impact of introducing program planning in a single lecture to the first-year computer science students. They discover that while introducing the concept of program planning in a single lecture may result in improvements in students’ planning behavior, students “need some computing experience before they can embrace planning” [22].

The latter is *How to Design Programs* (HtDP) [23] by *Program by Design* [24] project. HtDP is a course curriculum in *Racket* (a dialect of Scheme) designed around planning using a six-step design recipe process. By introducing the concept of planning with the design recipe early on and reinforcing it throughout the whole curriculum, this pedagogy guides beginning students to design, write, and test their programs more effectively. A significant portion of our work is based on our augmentation of the design recipe from HtDP and its benefits (presented in subsection 4.2.1) in combating some of the shortcoming we discuss in the following section.

2.5 Common Shortcomings of the Existing Alternative Methods

In this section, we highlight some shortcomings that are common to most of the existing alternative methods we explore in the preceding sections. We address

these shortcomings explicitly to distinguish the contributions of this thesis from the existing work on early computer science education.

2.5.1 Lack of Capitalization on Students' Existing *Programming* Skills

Many introductory computer science pedagogies are unique in a sense that they often lack a clear attempt to capitalize on the incoming students' existing knowledge or intuition on *programming*—something that might help the students connect or relate their previous experience to the new material to be presented. Perhaps this is due to the limited programming curricula that exist as discrete courses in the K-12 education standards. Therefore, it is understandable that many introductory computer science courses focus on starting mostly from scratch.

Nevertheless, we claim that just because the majority of incoming students have not heard much about programming, it does not mean that we must abandon all hopes for connecting some parts of computer science to something the students already know. We assert that presenting concepts without having any connection to the students' existing programming knowledge may make even some of the simplest concepts seem difficult, therefore attempts to make those connections whenever possible is pivotal in making early computer science education more approachable and effective. *Bootstrap*'s *Introduction to Programming* that explicitly instructs students to "use what you already know to think about" programming in Scheme, because it "works just like math" is an excellent example of this concept [25]. There may not be any particular programming course an introductory computer science instructor may be able to naturally reference in their first-day lectures like how a university's Calculus instructor may reference a high school AP

Calculus course. However, we find referencing practices from completely separate disciplines still proves useful. Further discussion regarding this is presented in section 3.3 and subsection 4.2.2.

2.5.2 The Cost of Increase in Cognitive Load

We argued that the extra cognitive load required to operate with the syntactic forms of specific programming language hinders students' ability to retain and express their algorithmic thought process. Some approaches like ILEs address this directly by reducing the learning curve of the syntax. However, we expect that the approaches that include additional pseudo-languages (section 2.1) or more programming languages (section 2.3) on top of the single programming language students already struggle with would not reduce the problematic cognitive load.

It is quite obvious why requiring students to learn multiple different languages would increase the cognitive load that hinders students' ability to focus on abstract fundamental concepts. However, we must better explain the logic behind why we claim that pseudo-languages do not help reduce the cognitive load.

Utilizing a pseudo-language that is free from syntactic rules enforced by a pedantic compiler or an interpreter may temporarily lower the students' cognitive load in externalizing algorithmic ideas. Nonetheless, many abstract languages and systems of pseudocode are designed to notate the application of challenging computer science concepts (such as nested iterations, recursion, and higher-order functions), and therefore use strikingly similar syntactic conventions as many formal programming languages do. This often leaves the lack of evaluators as the

only noticeable difference between the pseudo-languages and a formal programming language.

Simply taking away the mean-looking syntax error messages may help lower the students' cognitive load. However, even without the presence of those error messages, we must be careful not to neglect that the degree of *newness* a novice computer science experiences in an abstract language or pseudocode is often similar to that of any formal programming language. That is, students who have little knowledge of formal programming languages most likely also have little knowledge on any abstract languages or systems of pseudocode, if any. Thus, in order to meaningfully reduce the beginning students' cognitive load, we must utilize even higher level pseudo-languages. Allowing the pseudo-languages to take form of a natural language like regular English sentences would certainly free students from needing to learn some new elaborate syntax.

2.5.3 Persistence of the Blank Pages

Perhaps the most outstanding shortcomings of the existing alternative pedagogy is that it still leaves the beginning students vulnerable to "the Blank Page Syndrome," which Bloch et al. of *Program by Design* project identify as a phenomenon where "the student, given a problem statement, confronts a blank page...and doesn't know how to begin" [24]. Our observations from years of interacting with the students confirm that that this is a common shortcoming in many mainstream introductory pedagogies, including the alternatives we present in this chapter. We speculate the cause of this Blank Page Syndrome to be a two-fold issue.

First, beginning students are simply *too* new to the concept of programming. This connects to the point we make in subsection 2.5.1. There exists some pedagogy that alludes to the beginning students' existing knowledge, such as the aforementioned *Bootstrap* [25]. However, many mainstream pedagogies attempt to teach computer science as a mostly brand-new discipline without some connections to the students' past learning experience. This may unnecessarily amplify the learning curve and leave some students completely overwhelmed to even try doing *something*.

Second, on top the unprecedented learning curve they present, many pedagogies quite literally involve a blank page as a starting point of a students' programming environment. If the novice students' encounter of blank pages reliably causes them to feel overwhelmed and lost, why must the educators persist it in their pedagogies? We claim that as long as a programming environment lets a novice student encounter a blank page, its highly polished user experience and ease of use becomes a moot point. This may be analogous to being placed in front of a large blank canvas in an art studio. If you have limited knowledge on how to paint, you are not likely to be comforted just because the paintbrushes given to you are fancy or easy to use. We present our solution to this in subsection 4.2.1.

2.6 Contributions of This Thesis

With many of the work we have cited in this chapter as the inspirational foundations, we propose a new teaching method (presented in detail in Chapter 4). We identify the key contributions of this thesis as the following:

- Augmentation of the HtDP's design recipe for the effects-early languages.

- Proposal of a code outlining process to be combined with the augmented design recipe process.
- Peer review process of students' design recipe and code outlines.
- An experiment to validate the pedagogy we propose.

In the following chapter (Background), we present some key instruments that are prerequisites for establishing our definition of the new teaching method. Then in Chapter 4, we propose our new teaching method that addresses all shortcomings we enumerate in this chapter.

Chapter 3

BACKGROUND

In this chapter, we introduce a few key concepts that are instrumental in understanding the development of the new teaching method being proposed.

3.1 Design Recipe

Introduced by Felleisen et al. in *How to Design Programs* (discussed in 2.4, Planning-Based Pedagogy), design recipe is defined as “a roadmap for defining functions, which programmers use to make sure the code they write does what they want it to do” [26]. The original text presents the “basic steps of a function design recipe” as a six-step process including (1) data definition; (2) signature, purpose, statement, and header; (3) function examples; (4) function template; (5) function definition; and (6) testing [23].

```
; rocket-height : Number -> Number
; multiply the number of seconds by 7 to find the height
(EXAMPLE (rocket-height 11)
         (* 11 7))
(EXAMPLE (rocket-height 72)
         (* 72 7))
(define (rocket-height seconds)
         (* seconds 7))
```

Figure 3.1.1: Function definition written in Racket using the design recipe

In practice, following the six steps of the design recipe in a formal programming language results in the construction of function-level documentation comments (similar to *Javadoc* or *pydoc*), function headers, function bodies, and

(unit) tests for the functions written. Figure 3.1.1 above shows an example of this [26].

One theoretical benefit of using a design recipe as a part of early computer science education is that it can potentially address the “Blank Page Syndrome,” which we have pointed out as a common flaw of existing alternatives in Related Work (Chapter 2). By providing students with a concrete set of steps to follow, the design recipe provides them a viable alternative to blindly attempting “to piece together a program by trial-and-error” and skip the much-needed step to think through a given problem simply by saying, “‘Well, it seems to work...’ for what their program really does” [24].

In this thesis, we use an adaptation of Felleisen et al.’s design recipe in our development of a process that students are instructed to follow when expressing their function designs. Further illustration on developing this adaptation is presented in Chapter 4 (Proposal of a New Teaching Method), subsection 4.2.1 (Design Recipe).

3.2 Short-term Memory

In introductory computer science courses, we observe that students’ cognitive capabilities are often overwhelmed by the particularities of the syntax of a programming language they are not yet accustomed to, regardless of how simple the syntax may seem to the instructor.

In the field of cognitive psychology, the concept that the human mind can only hold seven plus-or-minus two items in its short-term memory has been a well-established fact since the publication of Miller’s seminal paper [27]. Since even the

elementary programming problems have a few steps involved in its solution (which a novice student may take three or four iterations of different approaches to arrive at), we theorize that this limited capacity of short-term memory is often fully utilized once a student is presented with a programming problem.

Our first-hand observations and experiences so far as students and educators in computer science substantiate this theory. As a student begins to engage with a given problem, their mind starts the process of constructing the initial road map on how to navigate the problem. At this point, the student may have developed some intuitions that could eventually lead them to a sound solution. However, once their focus shifts from abstract algorithmic thinking to the concrete implementation on a computer, the high cognitive load required to recall the unfamiliar syntax floods their short-term memory and interrupts the train of thought on the intuitions that were emerging.

Thus, proper externalization of the short-term memory prior to writing any code seems necessary to preserve the abstract algorithmic thinking process and any useful intuition that may result from such processes. In developing the new teaching method, we implement a mechanism that systematically introduces this externalization of short-term memory as part of the programming experience in an introductory computer science course.

3.3 Outlining

Software is a product of complex logic that is unique to the field of computer science, but computer science is not the only discipline that practices producing aggregations of complex logic. This suggests that computer science cannot be the

only field of study in which students suffer from the negative consequences of failing to properly externalize their initial roadmaps or intuitions on applications of complex logic. In fact, many non-technical disciplines today boast of robust strategies for organizing and externalizing complex logic, developed throughout the much longer history of their existence compared to the relatively brief history of computer science.

Arguably, one of the most prominent fields of study to have invested much effort in externalizing complex thought process is language arts. In composing a piece of writing that is to eventually span multiple pages, or even volumes, outlining is often utilized as the technique for externalizing the content of an author's short-term memory. Defined to be a practice of "identification of main ideas and supporting details ... and their representation in a specified format," this simple yet effective technique is "included in most elementary language arts curricula and is often taught" [28].

We take advantage of this well-established technique of outlining to develop the new teaching method, as the outlining borrowed from language arts provides one outstanding benefit: almost all students at higher level institutions are guaranteed to be familiar with the practice of outlining through the repeated exposure in their K-12 composition classes. Being able to utilize a technique with virtually no learning curve is ideal in attempting to minimize any factor that may impact the cognitive load of the students.

3.4 Nomenclature

In order to minimize any ambiguity in communicating our intent while presenting this work, we provide specific definitions for the following terms and phrases, to be applied strictly within the context of this thesis:

- **Formal Programming Language** refers to a general-purpose (as opposed to domain-specific), Turing-complete, and readily available programming language with an evaluator that allows such language to be executed on a computer.
- **Implementation Plan** refers to a set of steps to implement a particular solution to the given problem in a manner that can be used to compose an executable program in any formal programming language.
- **Friction** refers to a measure of students' emotional resistance to the teaching method being utilized.
- **The Framework** refers to the five key components of the new teaching method being developed and proposed in this thesis. The five key components are as follows:
    1. **Design Recipe** refers to our adaptations of the last five steps of the six-step process from *How to Design Programs* [23] [24], with augmentations to distinguish functional arguments from console and file system I/O's, as well as some formatting restrictions to allow for code template and unit test generation with a parsing tool.
    2. **Code Outlining** to externalize the abstract algorithmic intuitions in an easily readable and sharable format.

3. **Peer Review Process** for students to learn from each other and to practice effectively communicating different ideas for the given problem.
4. **Automatic Code Template Generation** to reduce some cognitive load spent on familiarizing oneself with the syntactic structure of a formal programming language.
5. **Automatic Unit Test Generation** to reduce the introductory-level learning curve to test-driven development while still encouraging thinking through the given problem sufficiently prior to implementation.

Further discussion on the specifications and the educational benefits of these components are presented in section 4.2, Components of the New Teaching Method.

- **Ability to Program** is defined as the ability for a student to adequately perform all of the following tasks:
    1. Effectively decompose a given problem into discrete subproblems and externalize them in the form of an implementation plan.
    2. Devise solutions to the subproblems in the implementation plan.
    3. Communicate the plans of implementation to other students.
    4. Devise a range of test values for the program-to-be-implemented.
    5. Follow the implementation plan and the test values to compose an executable program in a formal programming language that solves the given problem.

For the context of this thesis, **Ability to Program** is strictly limited to the application of *computer* programming.

Chapter 4

PROPOSAL OF A NEW TEACHING METHOD

In this chapter, we formally present the contribution of this thesis, which is a new teaching method for early computer science at higher education institutions. Strategies on integrating the new method into an existing introductory computer science course and its evaluations are presented in a later chapter (Chapter 5).

4.1 The Primary Goal of the New Teaching Method

In the previous chapters, we have argued that overreliance on the syntax of a formal programming language or languages is one of the critical issues that requires attention.

We must reiterate that our goal is not to oppose the use of formal programming languages and their syntax in early computer science education. Rather, we primarily intend to lighten the psychological burden placed on the novice students by providing a systematic approach they can utilize to effectively externalize and communicate their algorithmic thinking process. We expect this process to meaningfully reduce the necessity of having to recall syntax of a formal programming language when the students encounter any programming task in an introductory computer science course.

The new method we propose here has the ultimate goal of **improving the effectiveness of early computer science courses such that the participating students' ability to program is positively impacted in a quantitatively verifiable manner**.

4.2 Components of the New Teaching Method

The new teaching method we propose is centered around the framework, which contains five key components to be integrated into the existing curricula. The following subsections provide detailed specifications for each component of the framework, along with implementation examples that were used in validating the new method.

4.2.1 Design Recipe

Adapting and augmenting the last five steps (all steps excluding the data definition) of a six-step design recipe process from *How to Design Programs* [23] [24] (HtDP), we specify the design recipe as **the process of determining the name, inputs, outputs, purpose, side effects, and example test cases of a program, function, or a subroutine and explicitly externalizing them in a specific format**. The term itself ('design recipe') is also used to refer to **the textual artifact that results from this externalization process, such as a function documentation block** shown in Figure 4.2.1: Example of a design recipe as a textual artifact below.

```
"""
CONTRACT | updateAcceleration : float int -> float
-------: | :----------------------------------------------------------------
PURPOSE  | Given `gravity` and `fuel rate`, calculates the acceleration of the lunar module.
IN/OUTS  | None/None
EXAMPLE  | 1.62 5 -> 0     # constant velocity on the moon
         | 1.62 0 -> -1.62 # free fall on the moon
         | 1.62 9 -> 1.296 # max thrust on the moon
         | 9.82 7 -> 3.928 # strong thrust on the Earth
"""
```

Figure 4.2.1: Example of a design recipe as a textual artifact

As part of the framework, the process of constructing a design recipe is devised to serve as a streamlined tool for the students to effortlessly externalize their ideas on how to decompose a larger programming assignment into smaller pieces (namely, functions or subroutines). Naturally, this makes a few fundamental lessons—lessons on concepts regarding datatypes (such as Booleans, numbers, characters, and strings), functions or subroutines as smaller components of a larger program and strategies on unit testing—necessary prerequisites to the introduction of the design recipe process. Ideally, the first or second large programming assignment of the introductory course should be presented to the students along with the presentation of the design recipe process to highlight its application and usefulness. Provisions of a few concrete examples on how a programmer might propose their function designs using the design recipe process is also strongly encouraged.

Once the students are introduced to the concept of the design recipe process, they can be given an immediate opportunity to practice utilizing the process. For instance, the instructor may provide blank templates of the design recipe for students to use as they are coming up with the ideas on the functions they may want to build for the assignment. Electronic (Figure 4.2.2) and paper-based (Figure 4.2.3) examples of this template are shown in the figures below.

```
"""
CONTRACT | function-name : function-inputs -> function-outputs
--------:|:-------------------------------------------------------------------
PURPOSE  | purpose-statement
EFFECTS  | side-effect-in/side-effect-out
EXAMPLES | first-example     # optional-explanation-for-the-example
"""
```

Figure 4.2.2: An example of the electronic template for the design recipe

```
"""
CONTRACT | _____________________ : _____________________ -> _____________
PURPOSE  | ___________________________________________________________________
         | ___________________________________________________________________
EFFECTS  | None/None
EXAMPLES | _____________________________________ -> _____________________
         | _____________________________________ -> _____________________
"""
```

Figure 4.2.3: An example of a paper-based template for the design recipe

The primary benefit we expect from the design recipe process is almost identical to the motivation of the original design recipe proposed by the *Program by Design* project: “Addressing the ‘Blank Page Syndrome’” [24]. Unlike passing out a blank sheet of paper or instructing the students to open up a new text document, we expect that exposing the novice students to the template of the design recipe (such as the example shown in Figure 4.2.2 or Figure 4.2.3) will provide a strong sense of direction as to what they are supposed to do next.

More specifically, we expect that the blank fields of the design recipe sitting right in front of the novice students—almost asking to be filled in—are likely to organically lead them into coming up with a few concrete draft designs and test cases for the functions, effectively beginning to decompose the larger problem without having to think explicitly about problem decomposition as a task.

We believe this primary benefit of avoiding “Blank Page Syndrome” [24] is absolutely critical. No matter how great the framework may be, if the students feel unsure about what to do at the very beginning of practicing it, only to eventually get lead astray or give up, what value could it possibly deliver? Therefore, in defining the framework, we require the design recipe to be the very first component to be introduced to and practiced by the students, even at the instructor’s cost of having to spend a few days or weeks preparing the students with the fundamental concepts leading up to and including functions and unit testing.

As we discuss later, there are components of the framework (such as code outlining; subsection 4.2.2) that have almost no conceptual prerequisite and thus may be introduced to the students within the first few days of instruction. Nonetheless, our recommendation still stands with the design recipe having the highest topological ordering in relation to all other components, just so we can deliver the primary benefit of providing the students with a clear sense of direction.

As similar as it may seem to the original by Felleisen et al. [23], our adaptation of the design recipe process still has a few key differentiating features. The first differentiator is the aforementioned provision of the design recipe templates. (The original specifications of HtDP’s design recipe process do not require templates to be provided to the students [23], although there are implementations of it that do, e.g. *Bootstrap* [25]).

The second differentiator is that our design recipe process is completely agnostic to the specifics of a formal programming language’s syntax that may eventually be used to implement the function. Again, this is to reduce the possibility

of interrupting the students' thought process. Without the need to write the function signature in any formal programming language, the extra effort of having to recall the function header or signature syntax becomes eliminated from the process of problem decomposition and function design. Instead, students may simply state the name of the function, then list the datatypes of the functional arguments and return value in the `CONTRACT` line.

The third and final differentiating factor in our process is that the concept of side-effects is introduced from the very beginning, compared to how HtDP [23] defers introduction of side-effects and its inclusion in the design recipe until very late in its curriculum. Our design recipe process requires any side-effects of the functions to be explicitly stated separately from the functional arguments. This is motivated purely by our observation of novice students throughout the years of interacting with them: a recurring pattern we observe is that students tend to confuse and intermix the functional input/outputs and the system-level input/outputs such as the console outputs or keyboard inputs. Since the system-level input/outputs are often implemented as side-effects within a function, we expect to meaningfully reduce this confusion by letting students explicitly state the side-effects of a function and visually and conceptually distinguish any system-level interactions from the inputs and outputs of a function.

4.2.2 Code Outlining

Inspired by the techniques often taught and used in language arts [28], we define code outlining as **a process of describing the implementation plan for a program, function, or a subroutine**.

We want to minimize any extraneous cognitive load that may hinder the expression of the students' intuitive algorithmic thinking on how the programs, functions, or subroutines may be implemented. Therefore, code outlines are defined relatively broadly and informally. The outlines are to be written **in a natural language, in a bullet-point format, with optional utilization of different levels of bullets and indentation to distinguish the implementation plan within a control structure from the top-level implementation plans** (Figure 4.2.5). In theory, the outlines may be written in any medium, but we recommend instructing students to place the outlines as in-line comments in the source code file where the actual implementation in a formal programming language will eventually be. We make this recommendation to ensure that the efforts put towards composing the outlines contributes to the actual implementation when students begin to code.

When students design functions as part of the framework, we suggest that the code outlining practice immediately follow the design recipe process (presented in subsection 4.2.1), so that students may continue the externalization process of their intuitive ideas on functions as components of a larger program. We anticipate the practice of writing down or typing out the implementation plans in a concise format to assist the students in organizing, validating, and improving their initial ideas on how to implement certain componential algorithms of a larger program.

Because the code outlining process aims to capitalize on students' existing knowledge and experience with outlining in general, instructors need not invest much time into explaining how the outline shall be written. This also allows the

timing of integrating the code outlining process into the course curriculum to be flexible. Here, we provide a few examples of code outlines with varying complexity to illustrate this point.

Figure 4.2.4 shown below represents a potential externalization of a student's initial idea on how to structure a small top-down program as they are reading through the programming assignment specifications.

```
# get inputs for `skater's weight`, `distance`, and `object` to throw

# convert `skater's mass` to KG

# get `object's mass`

# get `object's velocity`

# get `skater's velocity`

# determine what `mass message` to print

# get the `skater's velocity announcement` message

# determine what `velocity message` to print

# print the messages
```

Figure 4.2.4: An example code outline for a simple program

In this example, the assignment specification asks for a simple program that computes the recoil velocity of a physical object (skater) based on the launch parameters (skater's weight, distance of the throw, and the type of a projectile) of a projectile thrown by it. Notice that the linear progression of logic in this outline could have been generated without any knowledge on even the most elementary computer science concepts such as functions or control structures. This suggests that outlines like the one shown could be generated by students of an introductory

course quite early on. To substantiate this, we provide a selection of linear code outlines (similar to Figure 4.2.4) written by actual introductory students in Appendix C, Selection of Student-Composed Linear Code Outlines.

With the limited complexity of the logic expressed in the outline, this externalization process may seem too trivial to provide any benefit. However, the benefit this seemingly trivial process provides is more than meets the eye. Unlike any retention of the initial idea in one's human memory, this once-externalized outline is semi-permanent, relatively free from the risk of being lost or corrupted [29]. Therefore, a novice student who has finished externalizing their initial idea for the program is then free to safely move on to other tasks required to complete the project. Continuing with the running example from Figure 4.2.4, perhaps a student may need to study some of the mathematical formulae to gain better insight on the domain knowledge (physics) required to understand the problem better, or they may desire to consider designing a function to handle detailed tasks like input validations (if the student is familiar with the necessary concepts, of course). Whichever the case may be, the novice student is now at a much lower risk of potentially losing a key insight from their initial brainstorming—for example, needing to perform a unit conversion to `KG` in the case of this program—than if they had relied on their memory alone.

Figure 4.2.5 shows an example of a code outline written for a 'word search' program, with the task of finding the locations of given words in a two-dimensional matrix of characters. Unlike the previous example, this outline is much larger in size, and it includes the aforementioned multi-level bullet points to account for the

applications of control structures such as conditionals and loops. Naturally, a student must have been introduced to the concepts of functions or subroutines and various control structures to be able to generate an outline like the one shown in the figure. Still, once the student has been taught those concepts, instructing them to express their implementation plan with those concepts in an outline format takes very little in-class overhead. Again, this is because we are able to capitalize on the students' existing knowledge and experience with outlining in general.

```
#reads string of letters from user and makes a list of list by using create_grid

#reads in second line of words from user into a list called words

#create an empty list to hold the results

#loop through each word in list of words

   # set rows equal to result of check_rows of word

   # set columns_down equal to result of check_columns_down of word

   # set columns_up to result of check_columns_up of word

   # set diagonal to result of check_diagonal

   #if rows at index 0 does not equal -1

      #then add rows to result list

   #else if columns_down at index 0 does not equal -1

      #then add columns_down to result list

   #else if columns_up at index 0 does not equal -1

      #then add columns_up to result list

   #else if diagonal at index 0 does not equal -1

      #then add diagonal to result list

   #else if none of the above work

      #add any one of the above results to result list (because every result will be a tuple with -1)

#loop through results list

   #if first term in an individual tuple in result list is -1

      #then print word and then 'word not found'

   #else

      #print each term in tuple formatted to fit instructors output
```

Figure 4.2.5: An example code outline for a moderately complex program

With a moderately complex program like the word search, expected benefits of externalizing the implementation plan as an outline become more apparent. First, an implementation of this size is arguably difficult to retain in one's memory without any form of externalization, so students shall benefit again by creating a semi-permanent record of their plan. Secondly, by writing the outline, students should

be able to visually organize the individual pieces of their plan such as strategic placements of control structures and function applications.

One may argue that a novice student is more likely to begin constructing their outline in a purely linear, top-down manner without necessarily thinking about applying any of the relatively complex concepts. We certainly anticipate some students may initially experience difficulties in developing an outline that goes beyond a linear, top-down approach, even for the problems with obvious opportunities to apply more complex logic such as conditionals and loops. However, as they continue the construction of their outline, we expect certain pieces of the outline to stand out as repetitive or particularly refactorable, perhaps motivating them to apply their knowledge of functions or control structures. Even if the original author of an overly simplistic linear outline does not catch certain opportunities to apply higher-order concepts, having a written outline makes pointing out such missed opportunities during a review process (such as the one presented in subsection 4.2.2) much easier.

Without any concrete artifact like the outline, motivating a constructive discussion of one's implementation plan would be rather difficult, as reviewing and critiquing something that only exists abstractly in one's mind usually is. In essence, this is the larger benefit we expect students to gain from practicing code outlining. Outlining is certainly a familiar practice that provides students with a tool to organize their intuitive algorithmic thought process. But more importantly, it yields semi-permanent and tangible artifacts which allows students to effectively revisit, review, critique, and further develop their algorithmic ideas with.

Figure 4.2.6 shows the code outline in a slightly different context, where the outline is used to externalize an implementation plan for a specific function rather than the top-level logic of the whole program. Here, the outline shown is constructed to accompany the design recipe for a function that searches for a possible occurrence of a `word` in a `grid` for the word search program, where `grid` is a two-dimensional matrix of characters.

Once the higher-level outlines and design recipes for their overall program are written, the only remaining portion of the implementation plan is the details on how the functions specified by the design recipes shall be realized. Therefore, we present the construction of more fine-grained implementation plan outlines like the one in the figure below as a natural subsequent step to the higher-level implementation outlines shown in the previous figures (Figure 4.2.4 and Figure 4.2.5).

```
"""
CONTRACT | check_columns_down : list str -> tuple
---------:|:-----------------------------------------------------------------
PURPOSE  | uses the `grid` and one `word` to check if word exists in column going down
EFFECTS  | none/none
EXAMPLES | [["HDFDS"] ["ESDFV"] ["LDSFA"] ["PSFCV"]] "HELP" -> ("HELP", "(DOWN)", 0, 0)
         | [["ABC"] ["DEF"] ["GHI"]] "BYE" -> (-1, "BYE")
         | [["LOW"] ["HOW"] ["IOW"]] "HI" -> ("HI", "(DOWN)", 1, 0)
         | [["PBZ"] ["PYZ"] ["PEC"]] "BYE" -> ("BYE", "(DOWN)", 0, 1)
         | [["BNY"] ["MKD"] ["DFN"]] "FUN" -> (-1, "FUN")
"""
# sets column index to 0

   # while column is less than length of rows of grid

      # create empty string called column

      # goes through each row in grid

         #add element in row at current column index of row to column string

      # goes through column string

         # if word is in column string use find to get index of word

            #return word, "(DOWN)", current index of word, and current column index as tuple

      # add one to column index

#return -1 and word as tuple
```

Figure 4.2.6: An example function-level code outline with a design recipe

Similar to how the assignment specifications may prompt students to think about the high-level solutions, we expect the design recipes to be good references for the students to base their function-level implementation plans on. We recommend the function-level outlines to be written adjacent to the design recipes, so students may conveniently refer to the contract, purpose, and examples of their functions to ensure that their outline is logically compliant with the design recipe.

4.2.3 The *DRaCO* Workflow

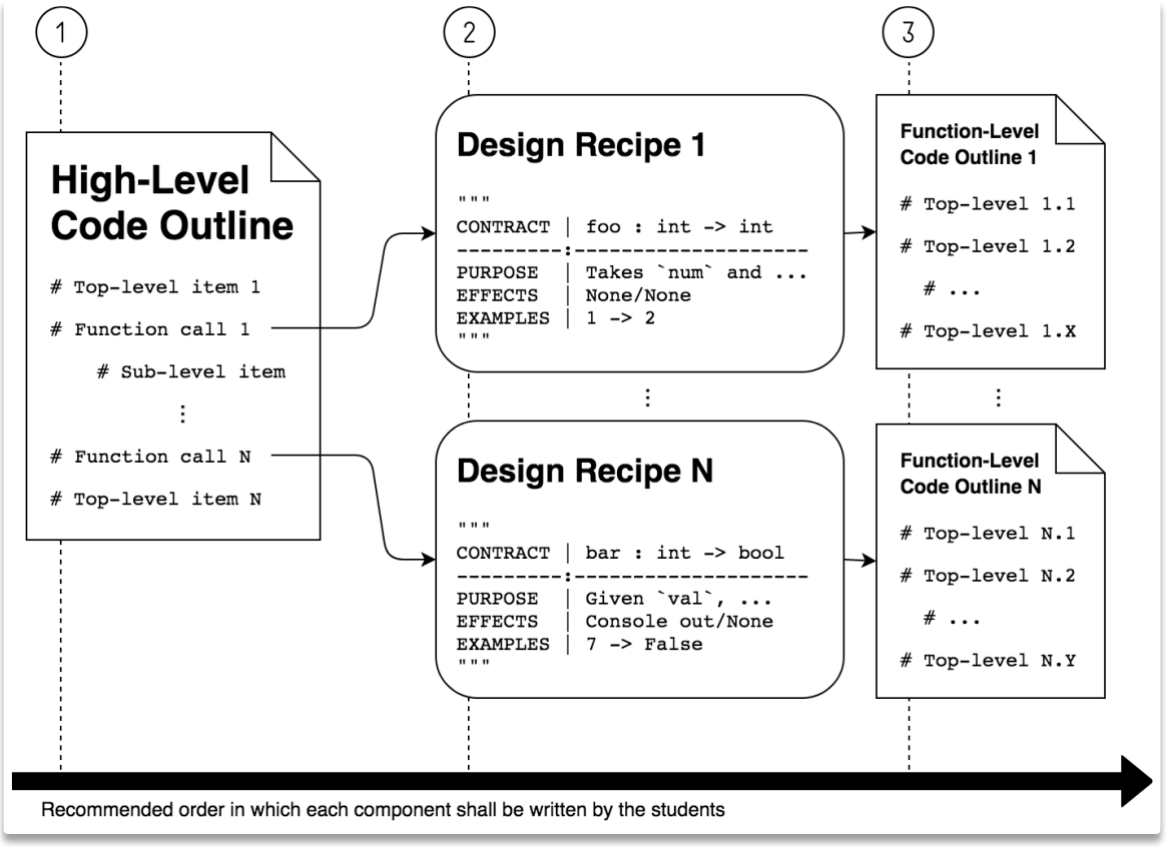


Figure 4.2.7: Design Recipe and Code Outline (DRaCO) workflow

Before we continue on to discuss the remaining components of the framework, we propose an aggregative terminology for the first two components presented so far: *DRaCO* (**D**esign **R**ecipe **a**nd **C**ode **O**utlines). Figure 4.2.7 above illustrates the process involving the two components (DRaCO) as a workflow.

We acknowledge that the aggregation of the two separate components of the framework as a single workflow may seem arbitrary and contrived. After all, the design recipe process and the code outlining practice each originate from different sources [23] [28] and have discrete purpose as we present in preceding sections. Nevertheless, we believe that it is important to present both as a singularly

packaged workflow to the students, since presentation of a single workflow is likely to minimize any room for confusion and make it more convenient when integrating it into an existing curriculum. Detailed discussion on implementing the DRaCO workflow into the existing assignment structures of an introductory course is presented in Chapter 5, Implementation of the DRaCO-based Pedagogy.

4.2.4 Peer Review Process

Peer code review is a process well known to be an effective tool both in the computing industry [30] and the educational context [31]. However, there are clear difficulties in implementing the process in the introductory courses. According to Busjahn et al, computer science educators report that "understanding the code's intention from the text surface" is perceived by the learners "as the major challenge in [code] reading" [32]. This is particularly easy to observe at the introductory level where encountering statements such as "I wrote this snippet of code, which does what I want, but I have no idea why it does what it does" are not a rarity. Many students struggle often with the code they have written all by themselves, so we can reasonably see how the magnitude of the struggles would only multiply if they had to read someone else's code.

This is perhaps similar to a situation in which a couple of students in a foreign language course—say, for instance, Korean—are attempting to communicate by speaking only in Korean. While it is a noble attempt on their end to practice the language, we can certainly expect some of their intentions to be 'lost in translation.' At least it makes sense for students in a Korean course to put themselves in a situation like this, as their primary goal in being in a Korean course

is to learn the language, not necessarily learning how to communicate their sentiments with each other.

Unfortunately, the same justification cannot be granted to the students who are taking an introductory computer science course. The primary goal of being in an introductory computer science course shall be, as we and many other educators have argued thus far [7] [6], anything but simply learning a programming language. Therefore, in most introductory courses, letting the algorithmic thought process get 'lost in translation' is in no way acceptable, since the part that often gets 'lost in translation' is what we care most about in a peer code review process.

Again, we approach this difficulty in integrating the peer code review process into an introductory course with recognition that the students' lack of proficiency in the formal programming language(s) is the main obstacle. We propose a peer review process that strips away the problematic "text surface" and retains only the "intention" [32], or the algorithmic thought process, behind. We claim that this is not too difficult to achieve with the specifications of DRaCO.

In the framework we are proposing, following the DRaCO workflow (Figure 4.2.7) results in generation of textual artifacts that are written mostly in natural language and formatted in a well-organized structure (bullet-point outline). The greatest benefit the DRaCO artifact delivers to the novice students is that almost all of the artifact is written in a language and format they are quite familiar with. If we assume that students have familiarized themselves with the formatting of design recipes (Figure 4.2.1), then we can even argue that none of the DRaCO artifacts shall seem foreign to them. By reading the code outlines that are

descriptions of different stages of the implementation plan in a natural language, students are effectively reading the original author's unaltered externalizations of the intentions. We expect this to be far more effective than having to roughly estimate the meaning behind a rather foreign surface syntax of a programming language that is possibly an incorrect translation of the author's original intentions.

Our specifications of the peer review process do not restrict the medium of the review. That is, the instructors implementing the peer review process as part of the framework are not at all discouraged from conducting the review with any electronic tools, including commercial code review tools such as Microsoft's *GitHub* [33], Atlassian's *Crucible* [34], JetBrains' *Upsource* [35], or any proprietary tools. Conducting the review simply with paper-and-pencil also aligns well with our specifications. However, we recommend that the review processes be held in person, such that the reviewer and the author of the DRaCO artifacts are free to verbally communicate while sharing a single copy of the artifact they are sharing.

Consequently, we define the peer review process in the context of the framework as **an in-person review process of the artifacts of DRaCO workflow, where the participants shall examine, discuss, and critique each other's implementation plans via verbal and written communication**.

Although we expect the peer review process without the overhead of having to translate the code to dramatically reduce the difficulty of communicating students' intentions, we also understand that it is quite unreasonable to expect all of the novice students to be perfectly proficient at clearly and concisely expressing their algorithmic thinking process even in a natural language. Given that many of

the students would be still learning how to think algorithmically with the concepts in computer science, we shall rather expect most DRaCO artifacts to lack clarity in many aspects. Since the main goal of the review process is to understand and critique the intentions of the author, we claim that having the author in the flesh to interact directly with is one of the most essential steps in achieving that goal. Especially if the artifact alone proves to be lacking due to the author's inexperience, the opportunities to freely ask clarifying questions, critique the logic expressed, and listen to the author's defense of certain choices would unequivocally add significant value to the review process than if the only channel of communication between the author and the reviewer were the DRaCO artifacts.

4.2.5 Automatic Code Template Generation

Primarily, both of the remaining components are included in the framework to dissuade students from simply perceiving the requirements of the framework as 'extra work' that is time consuming and offers little benefit.

The first of the two remaining components is automatic code template generation. Not to be confused with the design recipe template from subsection 4.2.1, we define 'code template' as **some starter code to be generated from the students' DRaCO artifact, such that the generated code may provide some guidance to the author of the DRaCO artifact on how one could start implementing their program in the syntax of a formal programming language**.

We observe that the interactive aspect of the formal programming languages is one of the key factors that captures students' attention and engagement. That is, students seem to enjoy observing that their code 'works.' It

is certainly understandable how a cycle of interaction between a programmer and a computer via a language that both can comprehend may be much more exciting than jotting down one's thought process in a manner that the computer will never understand. Although often overlooked in higher education, excitement is an important factor in a programming environment's ability to engage and retain students and their attention, as the developers of *Scratch* have noted in developing their initial learning environment for a wider range of audiences [14].

Therefore, we have devised automatic code template generation as one of the components of the framework in order to incite some excitement in students. The emphasis here is on the *automatic generation of code* based on students' text input. By providing a tool that parses DRaCO artifacts and generates some level of 'real code', we are allowing the emulation of the aforementioned cycle of interaction between a programmer and a computer. With this, we are also instilling into students' minds the idea that following the DRaCO workflow is more than mere notetaking, and that the DRaCO artifacts have some functional aspects similar to formal programming languages. With this, we aim to convince the students that DRaCO has a unique value beyond simply being a format to organize their implementation plan, thus motivating them further to put in sincere efforts into the DRaCO workflow.

We must note that we are being purposefully vague as to what the "generated code" or "some guidance" in the definition of a code template shall be. We acknowledge that there exist many schools of thought on where the line between beneficial and harmful lies when it comes to differing degrees of help a

code-generating tool may provide to novice students. While we still provide an example implementation of automatic code template generation in the subsequent chapter (Implementation of the DRaCO-based Pedagogy), we are leaving the interpretation of the definition of code template completely up to the discretion of the instructor implementing the framework to their own introductory computer science course.

4.2.6 Automatic Unit Test Generation

Automatic unit test generation in the context of the framework serves as the second component that adds practical benefit of utilizing the DRaCO workflow. It is quite similar to the automatic code template generation (subsection 4.2.5) in the sense that it also generates executable code based on the DRaCO artifacts and allows students to interact with a computer. However, automatic unit test generation is proposed with a much more clear-cut definition and purpose than providing “some guidance” to the students.

Defined as the **generation of a working unit test suite written in a formal programming language based on the DRaCO artifact**, we propose automatic unit test generation to serve as the training wheels for the novice students to experience the benefit of test-first or test-driven development (TDD).

Many years of research on TDD has revealed its positive and quantifiable impact on software quality, both in the industry [36] [37] and in academia [38]. However, as Edwards has pointed out, application of TDD is often met with challenges at an introductory level, because “software testing requires experience at programming,” and “introductory students are not ready for [software testing

practices] until they have mastered other basic skills" [39]. In addition, most TDD practices require writing the tests in the testing framework of the formal programing language in use. Since we are proposing an extended deference of any use of the formal programming languages based on the observation we have so far enumerated regarding our inexperienced target audience, it may seem that there exists no plausible avenue of compatibility between the traditional TDD practices and our Framework. While we acknowledge that no traditional TDD practice will organically synergize with the specifications of the framework, we also recognize that some logic to validate the students' design of the function or subroutines already exists in the form of example test cases within the design recipe component of the DRaCO artifact.

The key motivation here in specifying the automatic unit test generation is to utilize the validation logic already available in the form of example test cases to provide a working test suite the students can run as they begin writing code to implement their designs. One immediate consequence we may expect from this is that the students will proceed with a strong sense of direction as to what their implementation of the functions shall do, since running the generated test suite can provide instant feedback on how well their current code is working. This is precisely one of the major benefits of TDD [40]. Here, we are not claiming that the test suite generated from the novice students' DRaCO artifacts will offer the coverage level equivalent to the test suite written by an expert programmer. Our emphasis is on the fact that even a few runnable unit tests are far better at providing initial guidance than no unit tests, and that the whole test suite will be

bestowed upon the students at no cost of having to learn any particular testing framework or the language in which the framework is written.

Moreover, we anticipate that the students' initial interaction with a generated test suite will allow them to realize how a well-written set of test cases is capable of saving them from "forever fussing about what did I miss, what did I forget, what did I just screw up," as Kent Beck has put it [40]. With this, we assert that motivating the students to put more time and effort into writing good example test cases for the later assignments should not be difficult, especially with some extra guidance from the instructor to reinforce the importance of good test cases. Ultimately, we expect automatic unit test generation to at least cultivate a learning environment in which motivating and introducing more formally established practices of TDD becomes natural.

4.3 Summary of the Proposal

We present the entirety of our new teaching method with verbose philosophical and practical justifications in the earlier portion of the chapter. In this section, we tersely summarize our proposal and reproduce the definitions of the key components of the framework, such that those who intend to develop a proprietary integration strategy for the proposed teaching method may utilize it as a quick reference.

4.3.1.1 New DRaCO-Based Pedagogy for Introductory Computer Science Education

- The primary goal of the new pedagogy proposed here is to improve the effectiveness of early computer science courses such that the participating

students' ability to program is positively impacted in a quantitatively verifiable manner.

- The new teaching method is centered around the framework, which contains five key components. They are:
    1. **Design Recipe**: the process of determining the name, inputs, outputs, purpose, side effects, and example test cases of a program, function, or a subroutine and explicitly externalizing them in a specific format.
    2. **Code Outlining**: a process of describing the implementation plan for a program, function, or a subroutine in a natural language, in a bullet-point format, with optional utilization of different levels of bullets and indentations to distinguish the implementation plan within a control structure from the top-level implementation plans.
    3. **Peer Review Process**: an in-person review process of the artifacts of DRaCO workflow (subsection 4.2.3), where the participants shall examine, discuss, and critique each other's implementation plans via verbal and written communication.
    4. **Automatic Code Template Generation**: programmatic generation of some starter code based on the students' DRaCO artifact, such that the generated code may provide some guidance to the author of the DRaCO artifact on how one could start implementing their program in the syntax of a formal programming language.

5. **Automatic Unit Test Generation**: programmatic generation of a working unit test suite written in a formal programming language based on the DRaCO artifact.

4.4 Name of the New Teaching Method

In order to make the subsequent discussions clearer and avoid any confusion with other teaching methods, we refer to our newly proposed teaching method as a '**DRaCO-based pedagogy**' in the following chapters of this thesis.

Chapter 5

IMPLEMENTATION OF THE DRACO-BASED PEDAGOGY

In this chapter, we present the details of our implementations in integrating the DRaCO-based pedagogy into an introductory computer science course at California Polytechnic State University, San Luis Obispo ('Cal Poly SLO'). The proof-of-concept implementation of the framework presented in this chapter serves as a basis for our validation of the new pedagogy (presented in Chapter 6, Validation of the DRaCO-based Pedagogy) and is intended to be a motivating example for any replication studies or future work utilizing the framework.

5.1 Implementation Environment

We begin the illustration of the environment in which we deployed the DRaCO-based pedagogy by explaining the contextual course structures at Cal Poly SLO.

5.1.1 Introductory Computer Science Courses at Cal Poly SLO

At Cal Poly SLO, a majority of the underclassmen in computer science, software engineering, and computer engineering majors begin their major coursework in a sequence of two introductory courses: CPE-123 and CPE-101 (in this order). Each course spans a single academic term, which is a quarter composed of ten instruction weeks and a one final exam week.

The first course of the sequence, CPE-123, or *Introduction to Computing*, is a pre-introductory, "CS0" course designed to "to attract and retain undergraduates that have no prior experience in CS" [41]. This particular course is offered with

varying themes such as computational art, game development, mobile application development, music, robotics, and cybersecurity, in which a beginning student may choose to enroll depending on which subject aligns best with their interests. While all offerings of this course have a common high-level objective such as teaching "core computer science principles and tools, providing a foundation and context for more traditional, introductory CS coursework," the contents of the differently themed offerings of the course vary depending on the technology stack and the pedagogy employed by the instructor in charge [41].

The latter course in the sequence, CPE-101, or *Fundamentals of Computer Science*, is the traditional 'CS1' course. The university's catalog specifies the course to provide lessons on the following: "Basic principles of algorithmic problem solving and programming using methods of top-down design, stepwise refinement and procedural abstraction. Basic control structures, data types, and input/output. Introduction to the software development process: design, implementation, testing and documentation. The syntax and semantics of a modern programming language" (*Python 3* at the time of this research) [42].

We select the relatively traditional CPE-101 as a course suitable for introducing the DRaCO-based pedagogy, as it is a course with a well-defined set of objectives established over many years of refinement, and it is a course that satisfies the assumptions of our teaching method regarding the environments in which it may be implemented in. But, above all, it is a course that saw little change over the past half-decade in terms of how it has been implemented with a vehicle of a single formal programming language to convey the fundamental concepts.

5.1.2 General Structure of the CS1 Course at Cal Poly SLO

An offering of CPE-101 at Cal Poly SLO has a fifty-fifty split between the 'lecture' and 'lab' hours. With the current configuration of the course where a total of six hours is allotted to an offering of the course per week, three of the six hours are designed as a traditional lecture time during which the instructor of the course will deliver the course content to the students attending. Although most of the lecture hours are spent with the instructor presenting some computer science concept and its applications in front of a traditional classroom with little interaction among students, a small portion of the lecture hours are spent on discussions and opportunities in which students can work on small exercises as a group.

The remaining three hours per week are less structured and scheduled in a computer lab. During these 'lab' hours, students are encouraged to work on exercises and assignments for the course and seek the help from an instructor or a teaching assistant as needed. Rarely, some lab hours may be consumed by overflow lectures or exam time. Because the learning that is designed to occur during the lab hours are mostly student-led and not initiated by the instructor's delivery of new course material, attendance of the lab hours is largely considered optional.

Evaluation of the student performance in CPE-101 also follows the traditional classroom model. All students in computing-related majors, minors, or concentrations taking the course as a degree requirement are required to seek a letter grade ('A' through 'F') based on evaluation throughout the academic term. The letter grade for the course is computed largely based on the combination of

scores students earn from the following items: lab assignments (smaller programming homework to be completed within a short period of time; usually about a week), projects (larger programming assignments that require applying a sizable culmination of concepts, with relatively larger time window for students to work on them), a couple of midterm exams that occur throughout the quarter, and a final exam that is scheduled during the final exam week. *In principle*, all graded items in the course are to be completed without any collaboration among students.

5.1.3 Seams for the DRaCO-based Pedagogy

With the existing structure of the CPE-101 standing as presented in previous subsections, we identify the a few components of the course as 'seams,' or appropriate points in which we may be able to integrate the key components of the framework only with minimal and necessary disruption to the established course structure. These seams are described in the subsequent paragraphs.

First, we find the projects of CPE-101 to be perfect candidates for the students to apply the framework. The level of complexity the projects provide are deemed sufficient to motivate the need for thoughtful decomposition, planning, and testing prior to implementation in code. Also, the wide time window allocated for the projects allow equitable introduction of DRaCO workflow assignments and peer review process as smaller parts of the overall project progression.

Secondly, the midterm and the final exams provide ample opportunities for us to test the educational effect the framework has on students. Since the aim of the DRaCO-based pedagogy aligns well with the overall learning objective of the

course, we see that asking exam questions that are designed to test the students' ability to program would be appropriate.

Finally, the flexible lab hours that are scheduled in a separate time and physical location from the lectures do not only allow designing an A-B experiment to evaluate the pedagogy we were implementing (presented in Chapter 6) but are also pivotal in finding the extra time to introduce to the students key components of the framework and assist them in getting familiarized with the DRaCO workflow.

For the remainder of this chapter, we explain precisely how we integrate the components of the framework into the aforementioned seams on an offering of CPE-101 at Cal Poly SLO. We must note for clarification, however, that the integration techniques we present only apply to the experimental group of the course offering. That is, when we mention integration of certain components of the framework into our course offering of CPE-101, only half of the students (those in the experimental group) from our offering are affected. More details on the structure of the A-B experiment and the different course materials each of the experimental and control groups were exposed to are discussed in section 6.2, Experiment Design.

5.2 The DRaCO Workflow

Prior to assigning any actionable tasks pertaining to the DRaCO workflow to our students in CPE-101, we present the detailed implementations of individual components of the DRaCO workflow via instructor demonstrations and in-class discussions.

5.2.1 Implementation of the Design Recipe Process

Before students begin the design recipe process, we guide them to install the *Sublime Text 3* text editor. Then, we assist students in configuring its autocomplete feature to provide an electronic template for the design recipe—similar to the one shown in Figure 4.2.2. This configuration of *Sublime Text 3* text editor allows students to simply type '`dr`' in any Python source file and press the tab key on the keyboard to insert the design recipe template at the cursor location.

Once the template is inserted, subsequent presses of the tab key auto-advances the cursor and highlights different fields of the design recipe (function name, input argument types, return value type, purpose, and on) to be populated. Figure 5.2.1 is provided below as a snapshot amidst this process, which shows `CONTRACT` line of the design recipe populated, with the cursor auto-advanced prior to the completion of the `PURPOSE` line. Dashed texts such as '`purpose-statement`' are placeholder strings that get automatically highlighted and eventually replaced as the actual content of the design recipe is typed into the template as part of the process.

```
"""
CONTRACT | multiply_ints : int int -> int
--------:|:-----------------------------------------------------------------
PURPOSE  | purpose-statement
EFFECTS  | side-effect-in/side-effect-out
EXAMPLES | first-example     # optional-explanation-for-the-example
"""
```

Figure 5.2.1: Population of a design recipe template using auto-advance

Along with this setup to conveniently follow the design recipe process, a demonstration on how the design recipe assists in concretely specifying functions

are shown to the students via a few examples. Here, we reproduce those examples and explain a manner in which the students are expected to compose them.

```
"""
CONTRACT | print_hello : None -> None
--------:|:-----------------------------------------------------------------
PURPOSE  | Print "hello" to console.
EFFECTS  | None/str
EXAMPLES | None     # Function has no output!
"""

"""
CONTRACT | print_product : float float -> None
--------:|:-----------------------------------------------------------------
PURPOSE  | Given `a factor` and `another factor`, displays their product.
EFFECTS  | side-effect-in/str
EXAMPLES | None     # Function has no output!
"""
```

Figure 5.2.2: Example design recipes for functions with side-effects

Figure 5.2.2 above depicts the first two examples presented to the students as application of the design recipe process to specify functions that handle the console output. There are a few notable structural features shown in this example that were designed specifically to suit the projects in CPE-101. We begin by explaining the purpose of these structural features prior to discussing the conceptual features of this formatting.

The textual artifact of the design recipe process is encapsulated within Python's triple-quote (""") *docstring* comment enclosure and formatted to be *GitHub-style-Markdown*-compatible. Encapsulation within the Python docstring allows easy and proximate inclusion of design recipe into the source code later, allowing DRaCO to serve as a useful reference while students code, and as well-structured documentation once the implementation is completed. The GitHub-

style-Markdown compatibility enables programmatic transformation of the design recipe as a portion of a HTML document, similar to how Java's *Javadoc* comments can be compiled into pieces of API documentation in HTML.

Now, we dissect each line of this design recipe template and discuss the conceptual role each part of the line serves. First, the `CONTRACT` line requires specifications of the function's name, types of its input arguments, and the type of its return value, typed in the following format:

```
<function name> : <in1> <in2> … <inN> -> <return type>
```

The function name must be specified in a `CONTRACT` line, but the types of input arguments or the return type shall be stated as '`None`' if there aren't any.

The dashed line below the `CONTRACT` line is part of the GitHub-style-Markdown syntax that renders the whole block of text as a table. Immediately below it is the `PURPOSE` line that asks for a concise statement that expresses the purpose of the function being specified. In it, each input argument to the function shall be named in the same order as it appears in the `CONTRACT` line. These names of the arugments are to be enclosed in grave accent or 'backtick' characters (`) to distinguish them from the rest of the purpose statement. Since the purpose statement is to be written in a natural language, there is no restriction (such as not allowing spaces) on the naming of the arugments. This is shown on the second example depicted in Figure 5.2.2.

The first two lines (ignoring the Markdown syntax) of this design recipe template provide space to describe *what* the function shall do, leading the students to prompt themselves with questions such as 'What will this function need to

achieve the purpose stated?' and 'What would be an appropriate data type for this function to return once it has finished its work?'

Once these questions are answered, students then would encounter an opportunity to referesh their memory on how function inputs and outputs differ from the side-effects that implement system-level inputs and outputs. Formatting of the `EFFECTS` line requires separetely stating any system-level inputs and outputs. The example functions `print_hello` and `print_product` specified in Figure 5.2.2 both have console outputs, so '`str`' is written as the system-level output to be implemented with a side-effect later. The general formatting of the `EFFECTS` line is as follows:

```
<sytem-level inputs>/<system-level outputs>
```

Describing the system-level outputs in terms of the available datatypes is not critical on the `EFFECTS` line. However, use of terms coherent with the rest of the design recipe block is strongly encouraged.

The last line is `EXAMPLE(S)`, where students must write a few test cases for the function they have just specified in the first three lines of the design recipe. In order to reinforce the concept that functions with only side-effects (such as displaying result of a calculation to the console) and no return value (specified as '`-> None`') cannot be tested by inspecting the function's return value, examples in Figure 5.2.2 were shown with test cases as `None`. However, if the function being specified has a predictable output based on the inputs given, students are instructed to write a few example test cases as shown in Figure 4.2.2.

We acknowledge that our detailed, line-by-line description of the design recipe process may convey an image that the whole process is perhaps excruciatingly daunting or tedious to the students who are instructed to follow it. However, that is simply not the case. A live demonstration of the whole process takes minutes at most, with many students able to complete this process for a single function within a minute or two once they have had a chance to practice it a few times.

5.2.2 Implementation of the Code Outlining Process

As we underscore in our initial proposal of the code outlining process, code outlining aims to capitalize students' existing knowledge on the general concept of outlining. Therefore, unlike in implementing the design recipe process, we do not provide much specifics on how the outline shall be composed.

The only implementation detail on the process of outlining presented to students is that they must be written as Python's in-line comments, prepended by a hash (#) character instead of a rather-difficult-to-type bullet (•) character. For the function-level outlines, there is an additional directive to place the code outline immediately below the design recipe block.

Aside from the sheer convenience factor, this is to yet again ensure the easy and proximate inclusion of code outline directly into the source code, such that the code outlines may serve the dual purpose of being references at implementation time and being documentations post-implementation. Later, as we introduce control structures in class, a multi-level outlining technique is demonstrated in parallel as a suggestion for how the students may organize their

implementation plan more effetely with readability in mind. Figure 4.2.6, which is based on an actual DRaCO artifact composed by one of the students, is an excellent illustration of all details discussed here.

5.2.3 Integration of the Workflow

During the lecture, we introduce the design recipe process as a one that is discrete and narrowly-purposed. This introduction is presented immediately following the introductions of elementary concepts such as variables, expressions, primitive data types, and functions. The code outlining, on the contrary, is subtly presented throughout the early lectures of the course without explicitly drawing students' attention to it. Whenever the course material calls for writing a few lines of code as demonstrations, we start the whole process by writing a few lines of in-line comments as an outline for the code we are about to show.

With these preparations complete, the DRaCO workflow is then presented as a singularly packaged process that combines the design recipe process and the outlining practice. This combined workflow is included in CPE-101's project specifications in Appendix D. In those specs, we instruct the students to begin the design recipe process by creating a new Python source file with an extension `*.oln.py`, with '`oln`' being the abbreviation for 'outline' to indicate that the students are still composing the outline of the program rather than the actual implementation. Once the file is created, students may use the auto-completion feature of the text editor to recall and insert the design recipe template in an instant. The rest of the process is mostly compliant with the implementation details presented in the two previous subsections 5.2.1 and 5.2.2.

As the last step prior to the submission of DRaCO artifacts, the project specs guides students to create another Python source file with an extension `*.oln.py` to write their high-level code outline. This outline contains the implementation for the driver (`main` function) of the program. Ideally, with all of the functions involved in the project well-understood, students shall have little trouble piecing together the applications of their functions and completing the high-level outline.

A keen reader at this point may notice that this order violates the "recommended order in which each component shall be written by the students" suggested by the Figure 4.2.7 of subsection 4.2.3, The *DRaCO* Workflow. While it is true that instructing students to construct the design recipe prior to the high-level code outline conflicts with our original recommendation, specs for the first few projects of CPE-101 already include pre-designed functions for students to implement. This inclusion certainly takes away the students' freedom to decompose the large problem in a way they see fit. Nevertheless, we do not take any corrective measures to resolve this conflict, mainly for the reasons explained below.

First, we recognize that the pre-designed functions from CPE-101's project specifications provide an excellent example of good decomposition of a large problem, while also letting the students practice the process of transcribing the designs of functions presented in the project specs into DRaCO artifacts. These are both arguably desirable occurrences in the first two weeks of reinforcing the DRaCO workflow. Second, as we have mentioned previously, our priority lies with

minimal and necessary disruption of the existing curricula. We determine that altering a set of well-established project specifications to require deprecating part of the existing course infrastructure and evaluations rubrics merely for the sake of pedantic compliance to a recommendation of the pedagogy to be neither minimal nor necessary.

5.3 Peer Review Process

With every introduction of a project in CPE-101 that requires completion of DRaCO workflow, we set the due date of the DRaCO deliverables at least a week ahead of the final program deliverable due dates. This motivates students to complete the DRaCO workflow prior to the completion—or, ideally, the start—of actual implementation of the projects and leaves ample time for the instructors to pre-screen the artifacts and prepare for a review process while the project is still ongoing. Once DRaCO artifacts are collected on the due dates, instructors may pre-screen, format, and redistribute the artifacts. Printed artifacts are passed out to the students such that they are reviewed by a student of the same course other than the original author.

We present our implementation of the review process in the subsections below. First, we illustrate how the pre-screening and formatting process is applied to student submissions. Then, we explain two different strategies (informal in subsection 5.3.2 and ‘in-depth’ in subsection 5.3.3) we applied in conducting the reviews during the class times of CPE-101.

5.3.1 Pre-screening and Formatting

This first step of the review process is entirely dependent on the instructor. Although the specifications of the framework do not require DRaCO artifacts be pre-processed in any way by the instructor of the introductory course, we still claim that some level of inspection and correction can go a long way.

As the work produced by novice students usually is, we expect a fair number of mistakes such as incorrect formatting or typos be part of the deliverables. While completely tolerable and mostly harmless, we still deem those mistakes as potential distractions during the review process. We recognize that the novice students may find pointing out a grammatical mistake or a formatting error to be much more attractive option than having to criticize some potentially unsound logic during the reviews. Therefore, in our implementation, we visually inspect the student submissions to manually correct any non-semantic error, as long as our corrections do not alter the logical process the student demonstrates.

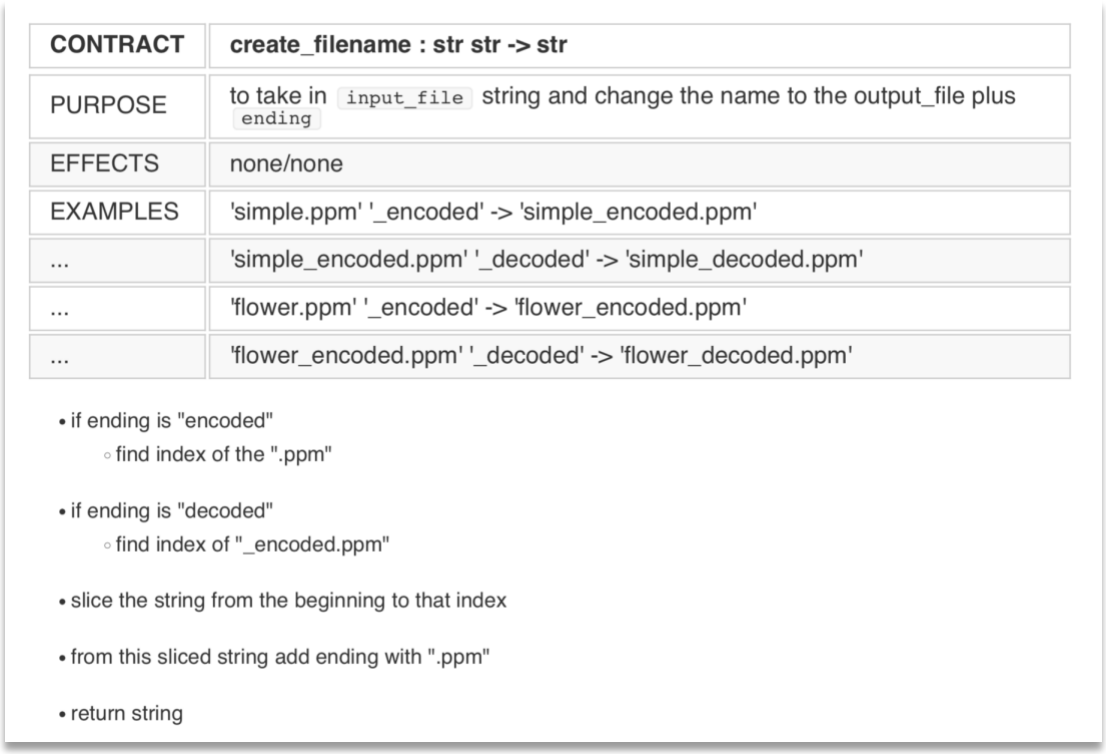
Once any outstanding non-semantic errors are corrected, we convert the Python source files (`*.oln.py`) to a HTML document and export the print layout as the PDF documents to make redistribution of the DRaCO artifacts more convenient for the instructors.

Figures below show the DRaCO artifact before and after the conversion process. Both figures depict the design of a function that processes a file name to generate a string to be used as another filename. Figure 5.3.1 shows the way a student initially constructed the design recipe and the code outlines, and Figure 5.3.2 shows the same artifacts once they has been fully converted.

At the end of the pre-screening and formatting step, we print out paper copies of the converted PDF documents in a random order so that they can be distributed to the students at the beginning of the review sessions.

```
"""
CONTRACT | create_filename : str str -> str
--------:|:-----------------------------------------------------------------
PURPOSE  | to take in `input_file` string and change the name to the output_file plus `ending`
EFFECTS  | none/none
EXAMPLES | 'simple.ppm' '_encoded' -> 'simple_encoded.ppm'
         | 'simple_encoded.ppm' '_decoded' -> 'simple_decoded.ppm'
         | 'flower.ppm' '_encoded' -> 'flower_encoded.ppm'
         | 'flower_encoded.ppm' '_decoded' -> 'flower_decoded.ppm'
"""
# if ending is "encoded"

    # find index of the ".ppm"

#if ending is "decoded"

    #find index of "_encoded.ppm"

# slice the string from the beginning to that index

# from this sliced string add ending with ".ppm"

# return string
```

Figure 5.3.1: DRaCO artifact in a Python source code file



CONTRACT	create_filename : str str -> str
PURPOSE	to take in `input_file` string and change the name to the output_file plus `ending`
EFFECTS	none/none
EXAMPLES	'simple.ppm' '_encoded' -> 'simple_encoded.ppm'
...	'simple_encoded.ppm' '_decoded' -> 'simple_decoded.ppm'
...	'flower.ppm' '_encoded' -> 'flower_encoded.ppm'
...	'flower_encoded.ppm' '_decoded' -> 'flower_decoded.ppm'

- if ending is "encoded"
  - find index of the ".ppm"
- if ending is "decoded"
  - find index of "_encoded.ppm"
- slice the string from the beginning to that index
- from this sliced string add ending with ".ppm"
- return string

Figure 5.3.2: DRaCO artifact in a PDF document after the conversion

5.3.2 Informal Peer Review Session

Despite the potential benefit [30] [31] of a well-structured code review, we do not consider possibly overwhelming the students with procedural details of a rigorous code review to be appropriate for an introductory phase of CPE-101.

Therefore, for the three out of five applicable projects in the course, we conduct the review sessions in an informal, student-led manner. Once we distribute the printed copies of the DRaCO artifacts and ensure that no student has their own artifact to review, we do not give much specific instruction other than that the students shall rearrange their seating such that the author and the reviewer can freely communicate.

By imposing very little procedural restrictions on the students, we allow them to move at their own pace, reading and comprehending the implementation for the same project their peer has constructed. Compared to a more formal review process with more explicit directions on which questions are to be asked and what features are to be discussed, this informal process may leave some students clueless as to how they shall critique the DRaCO artifact in front on them. Nevertheless, during the introductory phase of the course and the review process, we purposefully let students prioritize understanding someone else's logical thought process, rather than directing them to focus on certain points for the critique.

With this, we find many students asking for and listening to the explanations regarding certain algorithms from the author of the implementation plan. In some cases, small group of students form to discuss algorithmically challenging part of the project, with discussions involving how different strategies may be used to achieve the same goal. Despite the drawbacks of not providing detailed instructions, our observation of the student communications throughout the informal process to support our approach to still provide sufficient educational value.

5.3.3 In-depth Peer Review Session

For the latter two out of five applicable projects of CPE-101, we implement more rigorous review procedure with specific focus on what to review from the DRaCO artifact. We name this process as an 'in-depth' review.

Setup and the format of the code review (how students are each given someone else's implementation plan to review in person) largely remains identical to the informal process. One large difference for the in-depth process is that we require students to fill out an in-depth review worksheet, which asks specific questions.

We include questions on the worksheet that directs the reviewer's attention on how their peer implements certain features, how some corner cases for the projects are handled, and how easy it is to make out certain logical features from the artifacts. We also place some questions to encourage comparative analysis, asking for the reviewer to point out some logical similarities and differences between the reviewer's implementation plan and the implementation plan they are reviewing. Once the students complete the review and answer all questions on the sheet, we ask them to make a qualitative evaluation of the implementation plan reviewed as a whole. We do this by requesting an assignment of a letter grade ('A' through 'F') from the reviewer on the review sheet.

Due to the limited class time and many components of the DRaCO artifacts students must review, we observe that the in-depth reviews proceed with a sense of urgency, where the students' focus on the completion of the worksheet results in reduced volume of free discussions. Nevertheless, we also observe that the pinpointed critique questions posed by the worksheet reveals important parts of some implementation plan for some students. This allows some exchange of questions and answers among students regarding critical realizations about the

projects' implementations that might have not occurred during the student-led informal review process.

We provide examples of the in-depth review worksheet in Appendix E, In-Depth Code Review Worksheets.

5.4 Design Recipe and Code Outline Processor (DRCOP)

Our automatic code template and unit test generation tool, named 'Design Recipe and Code Outline Processor'—DRCOP for short (pronounced 'Doctor Cop')—serves as a proof-of-concept automatic code template and unit test generator specifically for CPE-101. In this section, we briefly present the scope of the tool and how students may use the tool as a part of the DRaCO-based pedagogy. Implementation-level details of DRCOP is presented separately in Appendix A, Architecture Design of DRCOP.

5.4.1 Scope of DRCOP

DRCOP parses students' function-level DRaCO artifacts (`*.oln.py` files) and generates the code template and unit tests in a correct Python 3 syntax.

Given the information present in students' artifacts, we can go as far as to provide control structure stubs and some snippets of function body using more elaborate techniques like keyword detection or natural language processing based on the function-level code outlines. However, since Python is already a high-level language with relatively terse syntax, we determine that the benefits of providing anything more than the function stubs would likely fail to outweigh any risk of potentially taking away some opportunities for students to practice writing Python

code. Thus, we purposefully limit the degree of ‘helpfulness’ in the generated starter code by providing only syntactically valid function stubs. Figure 5.4.1, which shows an unaltered output of DRCOP generated from the design recipe block in Figure 4.2.1, illustrates an example of this.

```
def updateAcceleration(gravity, fuel_rate):
    """
    CONTRACT | updateAcceleration : float int -> float
    PURPOSE  | Given gravity and fuel rate, calculates the acceleration of the lunar module.
    IN/OUTS  | None/None
    EXAMPLE  | 1.62 5 -> 0     # constant velocity on the moon
             | 1.62 0 -> -1.62 # free fall on the moon
             | 1.62 9 -> 1.296 # max thrust on the moon
             | 9.82 7 -> 3.928 # strong thrust on the Earth
    """
    pass    # delete "pass" once you have real code for this function!
```

Figure 5.4.1: Python function stub generated using DRCOP

As for the unit test generation, we find Python’s `unittest` module to involve concepts and syntactic particularities that seem to have little contribution to students’ learning of fundamental programming concepts. Requiring the understanding of object-oriented concepts and Python’s implementation of it—for instance, having to understand what the keyword `self` means—is one example of such particularities. In that light, we take the opposite approach from the code template generation and implement DRCOP to generate all of the unit testing code based on the `EXAMPLE(S)` line of the design recipe block.

As the consequence of this initial focus, we implement DRCOP to generate two Python files per a single, function-level `*.oln.py` file: (1) function stubs as the starter code and (2) a complete test suite compliant with Python3’s `unittest` module. With this, students can follow the standard workflow of TDD [40] without

having to implement any of the test cases themselves. They can start their iterative implementation and improvement cycle by first running the generated test file and seeing most of the test fail, then work towards the final '`OK`' message from the unit test driver by focusing on resolving each of the failures. As they complete the body of each function, they can re-run the tests to check whether their implementation complies with the behavior their tests specifies. Assuming the students invested a fair amount of time and effort into writing the `EXAMPLE(S)`, they can achieve "instant confidence" [40] in the code they have written with the test suite from DRCOP.

5.4.2 Usage Pattern of DRCOP

The course infrastructure of CPE-101 at Cal Poly relies heavily on the UNIX system the school provides, which allows convenient assignment deliverable collection and grading for the instructors while letting students learn the basics of the command-line UNIX environment. For instance, students are instructed to set up and use their UNIX accounts for developing and testing their lab assignments and projects, with the requirement to use the command-line utility `handin` to submit the final deliverables of every programming assignment.

We deploy DRCOP with this existing infrastructure in mind. Once students have finished drafting their DRaCO artifacts, they can upload their artifacts to the school's UNIX server and run DRCOP on their function-level DRaCO with the following command in shell which is available as a BASH script publicly listed on the instructor's UNIX account '`doryu`':

```
~$ ~doryu/services/DRCOP <filename>.oln.py
```

At first, students are almost guaranteed to run into different severities of `PARSE ERROR`s (examples of these are shown in Figure A.2 and Figure A.3, in Appendix A), which may require some revision on their end before proceeding. After a few cycles of running DRCOP, encountering errors, revising their DRaCO, and re-running DRCOP, we expect students to have a code template to begin their implementations of the course project, along with a fully functional unit test suite they can run from the very beginning. Output from DRCOP that a student may see at the point where DRCOP is writing out the generated file contents is shown in Figure 5.4.2 below.

```
File '../data/outputs/funcs.py' already exists.

    Overwriting it may result in PERMANENT LOSS of data!
    Is it really okay for DRCOP to overwrite it? [Y/n] y

Template file '../data/outputs/funcs.py' has been generated.

File '../data/outputs/funcs_tests.py' already exists.

    Overwriting it may result in PERMANENT LOSS of data!
    Is it really okay for DRCOP to overwrite it? [Y/n] y

Unittest file '../data/outputs/funcs_tests.py' has been generated.
```

Figure 5.4.2: Code template and unit test generation prompts of DRCOP

The full instructions that describe this usage pattern to the students of CPE-101 is provided in Appendix B, Instructions for DRCOP Usage.

Chapter 6

VALIDATION OF THE DRACO-BASED PEDAGOGY

As we mention in section 5.1, Implementation Environment, we deploy our own implementation of the DRaCO-based pedagogy in an attempt to empirically validate its effectiveness and identify any practical challenges in integrating the new methods. We design an A-B experiment with a control group we teach with conventional methods of CPE-101 and an experimental group with the deployment of DRaCO implementation illustrated in Chapter 5. Our primary focus here is to objectively measure and analyze any improvements to the students' ability to program attributable to the DRaCO-based pedagogy, with a secondary goal of observing students' emotional resistance to the integration of the pedagogy—namely, friction (as defined in Nomenclature).

6.1 Thesis Statement

Since we are interested in discovering whether the DRaCO-based pedagogy positively impacts students' ability to program, we utilize the single-tailed, two-sample t-test to determine the statistical significance of applying the DRaCO-based pedagogy.

In order to apply the statistical test to our empirical evaluation, we must clearly define the Null and Alternate hypotheses for the A-B experiment. The hypotheses are presented in the following subsections.

6.1.1 Null Hypothesis

Use of a teaching method consisting of Design Recipes, Code Outlining, and Peer Review practices backed by Automatic Code Template and Unit Test Generation (namely, the DRaCO-based pedagogy) **does not generate any difference or worsens** beginning students' performance on exam questions that test their ability to program.

6.1.2 Alternate Hypothesis

Use of a teaching method consisting of Design Recipes, Code Outlining, and Peer Review practices backed by Automatic Code Template and Unit Test Generation (namely, the DRaCO-based pedagogy) **does generate improvement** in beginning students' performance on exam questions that test their ability to program.

6.2 Experiment Design

Here, we present the high-level design of the experiment to provide some context of our experiment and disclose any relevant details that may impact evaluation of the results.

6.2.1 Subjects and Sample Selection

Subjects for the experiment are 34 students enrolled in a single offering of CPE-101 course during the Winter academic quarter of 2018 at Cal Poly SLO. Any student who wishes not to participate in the research as a subject is allowed to opt

out without any negative consequences to the course activities or their grades. All students of the course agreed to participate in our experiment. We consider the subjects of our experiment to be a representative sample of all first-year computing major students at Cal Poly SLO.

We conducted a customary *Prior Programming Experience* survey at the beginning of the academic quarter to find the following characteristics about the sample group: out of 34 subjects, only one subject reported having absolutely no programming experience, and another subject reporting to have never taken CPE-123. The majority of the subjects reported having less than a year of programming experience including CPE-123 (54.6%). Another partially overlapping majority out of 34 reported having no prior experience with Python (55.9%). Figures supporting these results are presented below.

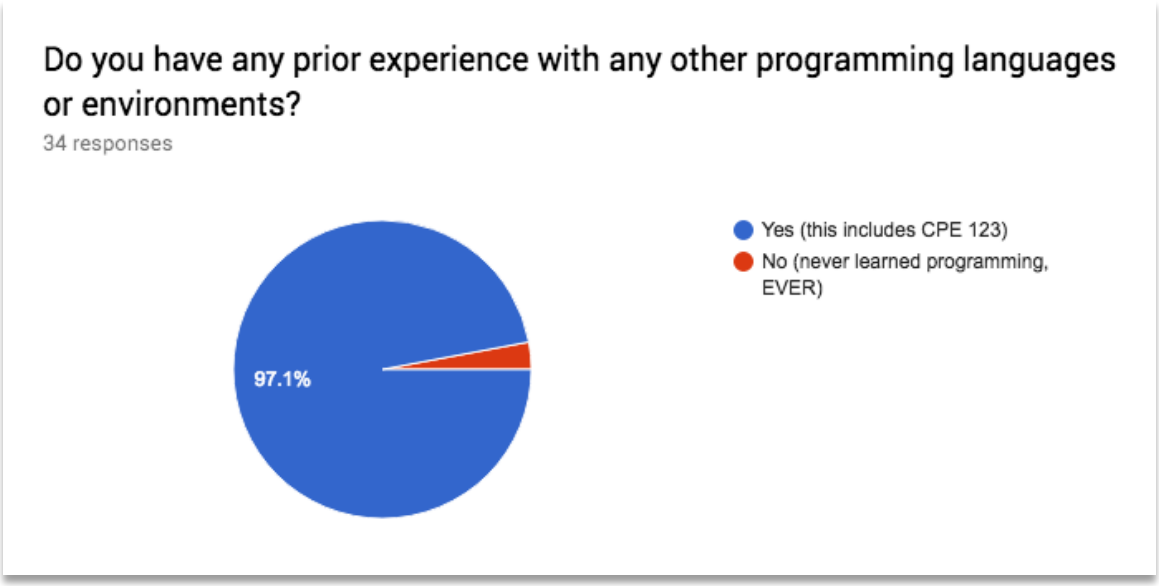


Figure 6.2.1: Subjects' experience in programming prior to CPE 101

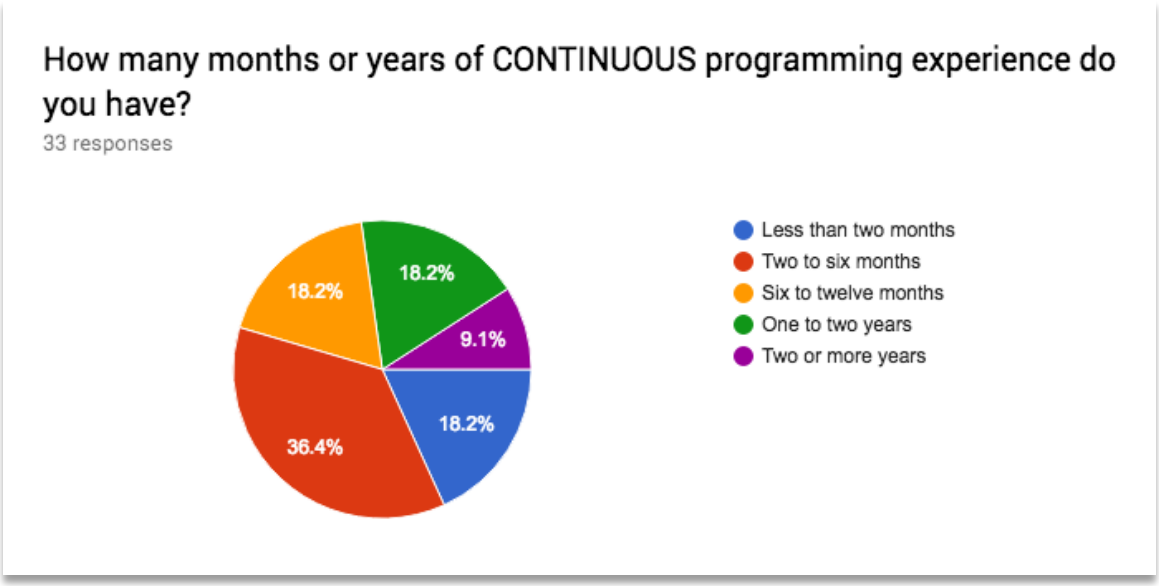


Figure 6.2.2: Durations of the subjects' prior programming experience

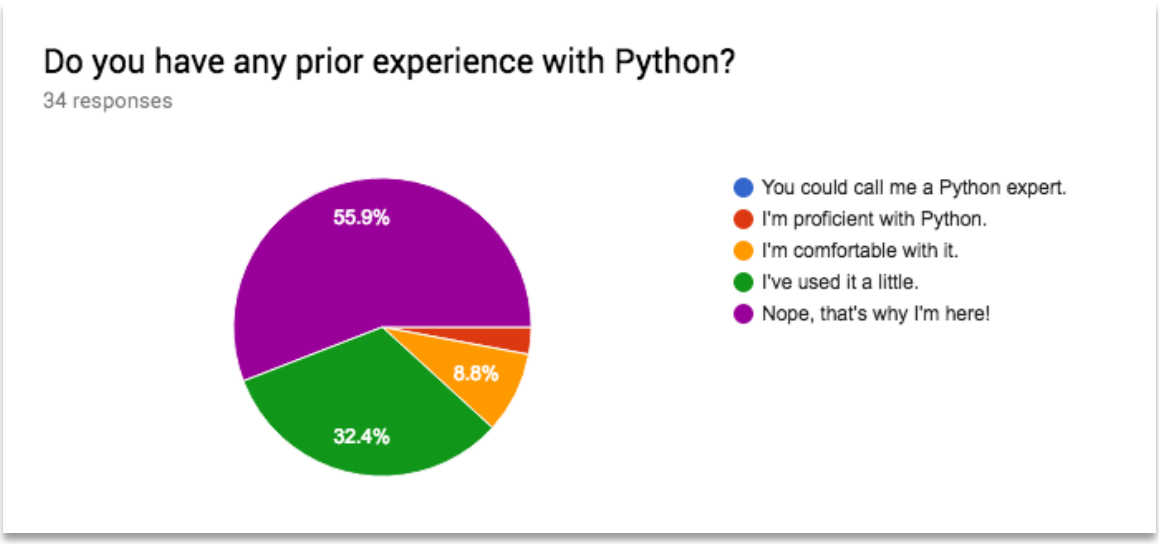


Figure 6.2.3: Subject's experience with Python prior to CPE-101

We leverage the separation of the 34 subjects into experimental and control groups on the lecture-lab split of CPE-101 course structure. Most offerings of CPE-101 at Cal Poly SLO maintain one-to-one mapping. That is, all students who attend a particular CPE-101's single lecture offering are assigned to a single offering of

the lab offering of the courses. However, we split the 34 students attending a single CPE-101 lecture by the primary investigator into two equally-sized groups of 17 students, each with their own lab offering to attend. The fifty-minute lecture of CPE-101 is scheduled on Mondays, Wednesdays, and Fridays at 12:10pm, with two separate fifty-minute labs scheduled back-to-back, one starting at 1:10pm (experimental group) and another starting at 2:10pm (control group). The splitting of the single group of lecture attendees into two groups of lab attendees are handled mostly by on the university's course enrollment system based on each students' other course schedules, preferences, and availabilities.

6.2.2 Execution

Throughout the ten-week duration of the quarter, two groups of students (control and experimental, each enrolled in different lab offerings) receive the same lectures, labs assignments, and projects, but are presented with different tools and procedures for the projects during the lab hours.

The control group is presented with the specifications that stress traditional methods of test-driven development with restricted peer evaluation or collaboration, whereas the experimental group receives instructions with heavy emphasis on practicing the DRaCO workflow, utilizing the textual artifacts of DRaCO to generate code template and unit tests using DRCOP, and participating in the peer review process.

6.3 Data Collection

The evaluation of student performance was done during the two midterm exams and a lab final exam at the end of the quarter. Two midterm exams—each scheduled during week four and six respectively out of a ten-week quarter—presented all students with the same problems, including a subset of problems designed to test their ability to program. The lab final exam was a computer-based exam where students were expected to complete a small programming task using the skills they have acquired throughout the quarter.

Reproduction of the subset of problems from the midterm exams and the lab final exam problem is available in Appendix F, Midterm and Lab Final Exam Problems . We summarize these problems and their design in Table 6.1 to show that our data collection methods comprehensively evaluate the DRaCO pedagogy's impact on students' learning.

In the table below, each problem is identified by the exam which it appears in and the problem number we assign to it. All problems are marked with number(s), one through five, corresponding to each component of the ability to program ('ATP') they are designed to test. The numbering scheme and the definitions for the components of the ATP are given in presented initially in section 3.4, Nomenclature. The rightmost column of the table explains the rationale behind each problem, i.e., why we believe it is important to gather our validation data based on it.

Table 6.1: Summary of exam problems that test students' ability to program

Exam	Problem #	ATP #	Explanation for the Problem Design
Midterm I	6	5	Tests students' ability to read and comprehend an implementation plan given in a code outline format, as well as their ability to correctly produce an executable program from the outline.
	7	5	Tests students' ability to distinguish the system-level input/outputs implemented as side-effect from the input/outputs of a function when implementing a function based on the provided code outline.
	8	4	Tests students' ability to generate effective test cases to cover all branches of a conditional logic. Also tests students' capabilities to extract and understand an abstract logic expressed in a formal programming language syntax.
	9	5	Tests students' ability to effectively comprehend and trace the execution pattern of an existing program. Also tests students' capabilities to navigate a complex iterative logic expressed in a formal programming language syntax.
Midterm II	8	1,3,4	Tests students' capabilities of understanding a complex problem statement and decomposing it into discrete subproblems in terms of function specifications in design recipe. Grading of this problem is done by a one-on-one interview, during which the authoring student is evaluated on their clear communication of their implementation plan.

	9	2,3,5	Tests student's ability to gauge the overall complexity of the functions they have specified as part of a decomposition process and their ability to lay out an implementation plan for them in terms of the code outline. Grading of this problem is also done by the one-on-one interview, during which the authoring student is evaluated on their clear communication of their implementation plan.
Lab Final	ALL	1,2,5	Comprehensive evaluation of a student's ability to program in real-life situation, where a student must decompose a complex problem into unit-testable functions, implement them, and utilize them effectively in a main driver program to satisfy the requirements for the given problem.

By collecting the scores earned on the exam problems by the two groups and analyzing the differences, we are able to validate if DRaCO-based pedagogy is effective in making a statistically significant difference in the experimental groups' ability to program. For the exam problems that have a set answer, we apply a rubric-based grading in evaluating students' responses to generate the scores, applying a class-wide rubric to ensure the grading is consistent throughout. For some exam problems that are more open-ended such as the ones asking for a decomposition of a relatively complex problem, we conduct a one-on-one student interview to ensure that the students' intentions are delivered clearly to the instructor while also testing student's ability to communicate their implementation plan to a third party (which is a component of the ability to program). Throughout

the grading process, we mix student submissions in random order and temporarily anonymize them such that no subconscious bias affects our evaluation.

In addition to the evaluation and analysis of exam results, in-person exit interviews and inspection of the student assignment submissions are performed to determine the students' emotional responses to the integration of the framework into the teaching of the course. This collection and analysis of student reactions is designed to reveal the magnitude of any friction introduced by integrating the framework into the course.

Although most of the data collected regarding friction is done informally via student interaction observations and making certain assumptions based on how students generate and deliver DRaCO artifacts, we do prepare a more formal interview process for the student exit interview, where we compose the following questions ahead of time and verbally deliver them to be answered by the students in a Likert scale, such that 1 indicates 'strongly disagree' and 5 indicates 'strongly agree,' unless otherwise specified. These questions are separated into the following sections for the better flow of the in-person interview:

1. Past Programming Practices
2. Newfound Programming Practices
3. Current Thoughts on DRaCO
4. Future Plans with DRaCO
5. General Feedback
6. Final Open Comment

These questions are presented in Table 6.2 below. Most questions are posed as statements students can respond with the Likert scale from ‘strongly disagree’ to ‘strongly agree,’ but questions that do not conform to this format are indicated with the curly brackets ({ }) following the question text that describes the different response type.

Table 6.2: Questions prepared for the student exit interview

Section	**#**	**Question** {Specifications of Different Response Type}
1	1	How much programming experience did you have prior to taking this course? {1: ‘Little to None’, 5: ‘Expert-Level’}
	2	Prior to taking this course, I have heard of or was taught code outlining or any other similar practice(s) for programming.
	3	Prior to taking this course, I was taught to use non-executable texts (such as in-line comments or paper-and-pencil notes) to plan out programming implementations.
	4	Prior to taking this course, I have had experiences of using a tool that generates code templates and/or unit tests from non-executable implementation plans.
	5	Prior to taking this course, I personally liked to write outlines for the program I am about to write before actually writing any executable code.
2	1	How much programming skill do you think you have now that you've taken this course? {1: ‘Little to None’, 5: ‘Expert-Level’}
	2	In this course, I have heard of or was taught code outlining or any other similar practice(s) for programming.
	3	In this course, I was taught to use non-executable texts (such as in-line comments or paper-and-pencil notes) to plan out programming implementations.
	4	In this course, I have had experiences of using a tool that generates code templates and/or unit tests from non-executable implementation plans.

	5	After my experience during this course, I now personally like to write outlines for the program I am about to write before actually writing any executable code.
3	1	I think constructing design recipe(s) is an important process in constructing a working program.
	2	I think writing code outline(s) is an important process in constructing a working program.
	3	I think peer DRaCO review process (would) have helped me better plan my implementation for a program.
	4	I think automatic function stub generation from DRaCO (would) have made my writing and completion of the programs easier.
	5	I think automatic unit test generation from DRaCO (would) have produced helpful unit tests I used throughout the course.
4	1	I plan to (continue to) use the DRaCO in my future courses if the automatic function stub and unit test generation becomes available for me to use.
	2	I plan to (continue to) use the DRaCO in my future courses, even if the automatic function stub and unit test generation is no longer available.
	3	I would recommend the DRaCO backed by automatic function stub and unit test generation to any beginning computer science student.
	4	I would recommend the DRaCO to any beginning computer science student, even without the support of any automatic function stub and unit test generation.
5	1	**If applicable**: I personally enjoyed writing design recipes for this course. {Empty response permitted}
	2	**If applicable**: I personally enjoyed writing code outlines for this course. {Empty response permitted}
	3	**If applicable**: I personally enjoyed using the function stub and unit test generation tool (DRCOP). {Empty response permitted}
6	1	How did you think the course went? {Free discussion}

	2	Lastly, do you have any questions, concerns, comments, or thoughts for the instructor? {Free discussion}

6.4 Experiment Schedule

Table 6.3 below enumerates each key component of the experiment we schedule throughout the course. In the schedule, we provide week and day numbers along with the concrete dates from 2018. The weeks are numbers 1 through 10, for each week of the ten-week academic quarter, and the days are numbered 1 for a Monday, 2 for a Wednesday, and 3 for a Friday class meeting.

Table 6.3: Experiment schedule in Cal Poly SLO's CPE 101

Date	Week	Day	Component of the Experiment
January 8	1	1	First Day, Informed Consent
January 17	2	2	Project 1 Assigned (no peer review)
January 26	3	3	Project 1 Due, Project 2 Assigned
January 31	4	2	Project 2 DRaCO Due, Informal Peer Review
February 2	4	3	Midterm I Exam
February 5	5	1	Project 2 Final
February 7	5	2	Project 3 Assigned
February 9	5	3	Project 3 DRaCO Due, Informal Peer Review
February 16	6	3	Project 3 Due
February 20	7	1	Project 4 Assigned
February 21	7	2	Midterm II Exam
February 23	7	3	Project 4 DRaCO Due, In-depth Peer Review
February 26	8	1	Project 4 Due
March 9	9	3	Project 6.1 Assigned
March 12	10	1	Project 6.1 DRaCO Due, In-depth Peer Review
March 16	10	3	Lab Final Exam
March 15 – March 22 (Finals)			Student Interviews for Midterm II and Friction

Chapter 7

RESULTS AND DISCUSSION

In this chapter, we provide the results of empirically evaluating DRaCO-based pedagogy with the A-B experiment we describe in the previous chapter (Validation of the DRaCO-based Pedagogy).

There are two main parts of the results of our experiment. The first is the statistical significance test based on the exam scores that reflect students' ability to program—namely, *Composite ATP Score*. The second is the evaluation of Friction based on the primary investigator's observation of student interactions during the academic quarter and students' responses from the exit interviews at the conclusion of the experiment.

7.1 Composite ATP Score

We compute composite ATP score per student, where a single score is calculated as a weighted mean of the scores earned on the specific problems that test their ability to program from Table 6.1: Summary of exam problems that test students' ability to program. The weight used for each problem is based on the specific problem score's contribution to the total course grade. Once the weighted mean is computed, we scale it as a percentage out of the maximum attainable score. Table 7.1 below shows an example of how a single student's composite ATP score is computed.

Table 7.1: Example computation of a single Composite ATP Score

Exam	Problem #	Score Earned	Max Score	Appx Score Weight

Midterm I	6	6	10	2.31 %
	7	5	5	1.15 %
	8	5	5	1.15 %
	9	6	6	1.39 %
Midterm I Subtotal		**22**	**26**	**6.00 %**
Midterm II	8	10	12	2.77 %
	9	4	6	1.38 %
Midterm II Subtotal		**14**	**18**	**4.15 %**
Lab Final	Output Diff	0	15	0.45 %
	Main	15	25	0.75 %
	Functions	24	40	1.20 %
	I/O	20	20	0.60 %
Lab Final Subtotal		**59**	**100**	**3.00 %**
Composite ATP Score Relative to Course Grade				13.15 %
Final Composite ATP Score for the Student				**76.61 / 100.00**

Table 7.2 below shows composite ATP scores for all students who were subjects of our experiment. Students from the experimental groups are marked with subject IDs ‘A01’ through ‘A17’, and the ones from the control groups are marked with subject IDs ‘B01’ through ‘B17.’ Values shown in this table are used in deriving the statistical significance (subsection 7.1.1) of our experiment.

Table 7.2: Composite ATP scores for all subjects of the experiment

Subject ID	Composite ATP Score	Subject ID	Composite ATP Score
A01	77.82	B01	43.75
A02	58.79	B02	78.05
A03	76.61	B03	71.96
A04	66.11	B04	81.88
A05	79.69	B05	47.26
A06	71.07	B06	74.18
A07	59.16	B07	69.12
A08	77.51	B08	69.28

A09	83.53	B09	29.82
A10	81.63	B10	67.11
A11	67.16	B11	70.51
A12	92.43	B12	80.56
A13	60.00	B13	45.42
A14	66.84	B14	41.56
A15	72.61	B15	50.70
A16	65.40	B16	63.65
A17	43.23	B17	84.46

We present the group means and the overall sample standard deviation of the composite ATP scores in Table 7.3 below. Values shown in this table are used in deriving the effect size (subsection 7.1.2) of our experiment.

Table 7.3: Composite ATP score statistics for the effect size computation

Means of Composite ATP Scores	Experimental (A)	70.56
	Control (B)	62.90
Sample Standard Deviation of All Composite ATP Scores	14.60	

Lastly, Figure 7.1.1 shows the kernel distribution plots of the two groups' composite ATP scores.

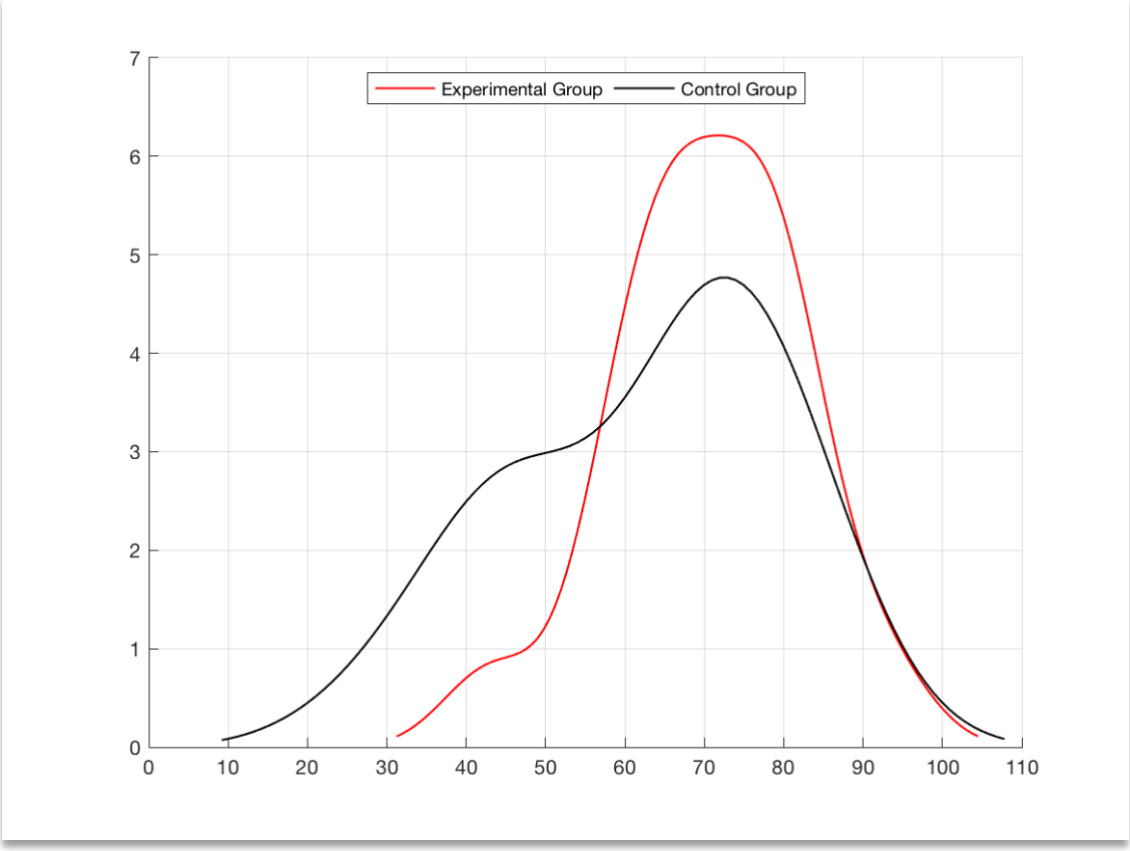


Figure 7.1.1 Kernel distribution plot of composite ATP scores

7.1.1 Statistical Significance

As explained in 6.1, Thesis Statement, we perform a single-tailed, two-sample t-test to determine the statistical significance in applying the DRaCO-based pedagogy to improve introductory computer science students' ability to program.

First, we begin by defining the parameters of the t-test. As for the variable to analyze, we select composite ATP score as the single variable for statistical significance test. For the significance level, we use the value pre-determined prior to the commencement of the experiment. This significance level, or Alpha (α) value, is set to 0.05. By using a pre-set significance level, we suppress any potential of

post-experiment alteration of the parameters of statistical analysis in order to fit the results to the hypothesis.

Our t-test returns the p-value of **0.0644** (rounded to the nearest ten-thousandth) which is larger than our pre-set Alpha value of 0.05. Therefore, we are not able to conclude that DRaCO-based pedagogy yields any statistically significant difference in students' ability to program.

7.1.2 Effect Size

The failure to show the statistical significance of implementing DRaCO-based pedagogy is not particularly surprising, given the total sample size of 34. What is surprising, however, is that the p-value we derived is close of enough to suggest at least some effect that DRaCO-based pedagogy has. In order to further investigate the effect of our new pedagogy, we compute the effect size. Adapting Coe's definition that "effect size is just the standardised [*sic*] mean difference between the two groups" [43] directly, we use the following formula to compute the effect size:

$$\textbf{Effect Size} = \frac{(\text{Mean of Experimal ATP Scores}) - (\text{Mean of Control ATP Scores})}{(\text{Sample Standard Deviation of All Composite ATP Scores})}$$

Using the values of means and the standard deviation presented in Table 7.3, we get the effect size of **0.5248** (again, rounded to the nearest ten-thousandth), which qualifies as "medium" effect size, and is "large enough to be visible to the naked eye" [43].

This result is certainly much more optimistic for the future of DRaCO-based pedagogy. Although we acknowledge that there are reputable criticisms of using

the term such as ‘medium’ out of contexts [44] [45], we retain our conclusion of ‘medium’ effect size based on Coe’s statement that “In education, if it could be shown that making a small and inexpensive change would raise academic achievement by an effect size of even as little as 0.1, then this could be a very significant improvement, particularly if the improvement applied uniformly to all students, and even more so if the effect were cumulative over time” [43].

7.2 Friction

Rather than applying rigorous statistical utilities to the data we have collected, we rely mostly on the qualitative observations and the experiences we have throughout the experiment to discuss Friction, supplemented by the students’ responses from the standardized exit interview. The questions we utilize in the exit interview are shown in Table 6.2: Questions prepared for the student exit interview. The entirety of the exit interview results in tabular and distribution plot formats is available in Appendix G, Student Exit Interview Results.

Above all, it becomes obvious quite early on that the integration of DRaCO workflow into CPE-101’s project specifications as a required deliverable appears as burdensome to the students, as it inevitably introduced additional intermediate deadlines and more submission requirements. We, as instructors, make an argument that front-loading the workload on the DRaCO workflow has a great potential to ultimately reduce any unexpected semantic errors later on, along with the added benefit of DRCOP’s automatic unit test generation from DRaCO. However, our observation shows that the long-term benefit of the DRaCO workflow is often eclipsed by the nuisance of having to deal with extra deadlines and

deliverables in the students' eyes. Experiencing student pushbacks such as requests to be exempt from the DRaCO requirements or pleading how DRaCO workflow is becoming an impedance for certain projects is quite a regular occurrence at the introductory phase. However, these pushbacks soon subside as we communicated to the students that the DRaCO requirements cannot be waived for any reason and students simply accepted as part of the course requirements.

Despite this pushback, we observe that a small group of students soon learn to take advantage of the DRaCO workflow heavily to their benefit, acknowledging that spending sufficient time on the DRaCO artifacts prior to the starting code implementation is indeed helpful in the long term. We find that almost all DRaCO deliverables we inspect from this group of students do support their claim regarding how much time and effort they spend on DRaCO.

We also observe that our implementation of automatic code template and unit test generation, namely DRCOP, seems to introduce a bit of extra difficulty for the students. Although we design DRCOP to be as user-friendly as possible, the limitations of command-line interface and the students' lack of familiarity with the UNIX environment help paint DRCOP as any other regular UNIX-style utility that is difficult to work with in many cases. While our exit interview results indicate that DRCOP still serves its purpose as a good motivator for students to use the DRaCO workflow on their own (delta between questions 1 and 2 from section 4, increase of 0.9444 in experimental group compared to the increase of 0.5000 in the control group on average), and that students certainly found the generated unit tests to be helpful (questions 5 from section 3, average response 4.2778), we encounter

confusions from students regarding their interaction with DRCOP quite often throughout the experiment.

Another notable observation includes the peer review process, where the exit interview responses (question 3 from section 3) reveal that students from the experimental group have rather lukewarm responses (average of 3.1111) regarding the peer review process's helpfulness, compared to the relatively high expectations the control group show for the same process (average of 4.0625).

Lastly, we use *Google Cloud Platform* (GCP)'s natural language sentiment analysis demonstration tool to analyze the overall sentiment captured during the final open comment section of the exit interview (section 6) [46]. The sentiment analysis result provided by GCP is composed of a sentiment score ranging between –1 and 1, accompanied by the magnitude of the sentiment as a positive real number. Detailed definitions of the output values are provided by GCP as follows [47]:

- `score` of the sentiment ranges between `-1.0` (negative) and `1.0` (positive) and corresponds to the overall emotional leaning of the text.
- `magnitude` indicates the overall strength of emotion (both positive and negative) within the given text, between `0.0` and `+inf`. Unlike `score`, `magnitude` is not normalized; each expression of emotion within the text (both positive and negative) contributes to the text's `magnitude` (so longer text blocks may have greater magnitudes).

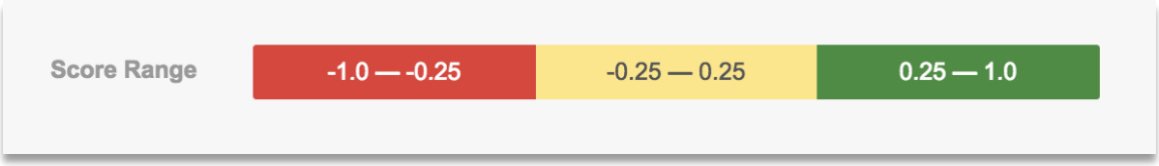


Figure 7.2.1 Google Cloud Platform's classification of sentiment scores

Figure 7.2.1 above shows the classification of the sentiment scores as negative (red), neutral (yellow), and positive (green). This analysis reveals that the experimental group's comments have an overall neutral sentiment about the course (sentiment score 0.2) at a relatively higher magnitude (33.9, 0.0209 per word), while the control group's comments contain an overall positive sentiment (score 0.3, 0.0392 per word) at a lower magnitude (28.3). If we assume that the difference between the two sets of values are larger than the margin of error, we may interpret this result as a supporting measure for the pushback and the difficulties we observed while deploying DRaCO workflow and DRCOP.

7.3 Overall Evaluation of the Thesis Statement

Considering the statistical and anecdotal analysis presented in previous sections, we determine that the use of a teaching method consisting of Design Recipes, Code Outlining, and Peer Review practices backed by Automatic Code Template and Unit Test Generation (namely, the DRaCO-based pedagogy) **shows promising potential** in generating meaningful improvement in beginning students' ability to program.

With this evaluation of the thesis statement, we also stress the implementation strategy we have utilized in conducting our experiment incurs at least some nontrivial Friction, such that the students working under our particular

implementation are likely to have negative emotional response to the DRaCO-based pedagogy. This indicates that at least a moderate amount of modification to our implementations of the pedagogy is required for a successful integration of the new pedagogy into existing curricula.

Chapter 8

THREATS TO VALIDITY

The primary threat to our work's validity lies in our methods of validating the new pedagogy we propose. First, our design of the exam problems and assignment of the score distribution are not particularly compliant with any existing standardized academic testing framework. Due to this, any lack of experience or knowledge we have from the field of education in general may have resulted in the exam problems that may not have been as effective in testing students' ability to program as we intended them to be. Second, although we have taken all cautionary measures necessary to prevent biased evaluation of the exam problems, we acknowledge that we were still susceptible to subconscious bias that may skew the outcome. This is almost unavoidable as we were both the author and the tester for the DRaCO-based pedagogy and its implementations.

Another threat to validity is our sample selection. The students who participated in our experiment were divided initially by Cal Poly SLO's course enrollment into the two separate lab offerings which we utilized as experimental and control groups. In addition, before commencing the experiment, we moved some students from the experimental-group-to-be lab offerings to the control-group-to-be lab offering to resolve the group size imbalance. We recognize this process does not follow proper random sample selection and division procedure.

Lastly, we note that students' own bias for or against the DRaCO-based pedagogy might have affected our validation, as our experiment was not a 'blind' A-B experiment. That is, all students in the experiment were enrolled in the

common lecture offering, and we did not restrict communications between the two groups of students. Most students were aware of how the other group of students were taught during the lab hours. Replication of our validation in a blind A-B experiment may yield significantly different results from the ones we achieved from our validation methods.

Chapter 9

CONCLUSION AND FUTURE WORK

9.1 Concluding Remarks

In this thesis, we motivate and propose a new teaching method for the early computer science education at higher education institutions. We use our own experience with observing students' struggles with the syntax of programming languages and the "Blank Page Syndrome" [24] to suggest a DRaCO-based pedagogy that capitalizes on fundamental programming skills the students possess as intuitions and their familiarity with outlining practices.

We develop our own implementation of this pedagogy to integrate it into Cal Poly SLO's introductory computer science course that. We set up an A-B experiment to validate the pedagogy and its implementation while observing the emotional response from the students regarding the new workflow and course requirements that we introduce on top of the existing curricula. We collect students' exam scores to verify that the new pedagogy shows strong potential in generating meaningful improvement in beginning students' ability to program despite some indications of negative emotional response from the students.

9.2 Future Work

The work we present here is highly experimental and is in need of further validation via replication studies that deploy improved implementations of the DRaCO workflow and the automatic code template and unit test generation.

If the pedagogy itself is shown to be sound with replication studies, the next step of development must be focused heavily on a streamlined implementation of the automatic code template and unit test generation. In particular, we anticipate a graphical tool that provides the same functionality of DRCOP in a much more streamlined and easy-to-use way to be one of the more attainable ways in significantly reducing Friction we observe in our experiment.

BIBLIOGRAPHY

[1] J. O. Shallit, "A Very Brief History of Computer Science," University of Waterloo, [Online]. Available: https://cs.uwaterloo.ca/~shallit/Courses/134/history.html. [Accessed 2 May 2018].

[2] U.S. Bureau of Labor Statistics, "Software Developers: Occupational Outlook Handbook," U.S. Bureau of Labor Statistics, 2017. [Online]. Available: https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm. [Accessed 2 May 2018].

[3] D. S. Alberts and D. S. Papp, "The information age: An anthology on its impact and consequences," Office of the Assistant Secretary of Defense Washington DC Command and Control Research Program (CCRP), Washington DC, 1997.

[4] M. Resnick, J. Maloney, A. Monroy-Hernàndez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman and others, "Scratch: programming for all," *Communications of the ACM,* vol. 52, no. 11, pp. 60-67, 2009.

[5] Lifelong Kindergarten Group at the MIT Media Lab, "Scratch - Imagine, Program, Share," Lifelong Kindergarten Group at the MIT Media Lab, 2018. [Online]. Available: https://scratch.mit.edu/. [Accessed 2 May 2018].

[6] S. Bloch, "Teach Scheme, reach Java: introducing object-oriented programming without drowning in syntax," *Journal of Computing Sciences in Colleges,* vol. 23, no. 5, pp. 65-67, 2008.

[7] S. Fincher, "What are we doing when we teach programming?," in *Frontiers in Education Conference, 1999. FIE'99. 29th Annual*, San Juan, 1999.

[8] R. Bornat, Programming from First Principles, Prentice Hall International, 1987.

[9] P. J. Landin, "The next 700 programming languages," *Communications of the ACM,* vol. 9, no. 3, pp. 157-166, 1966.

[10] R. Shackelford, Introduction to Computing and Algorithms, Addison-Wesley, 1998.

[11] S. Fincher, S. Cooper, M. Kölling and I. Utting, "ILE-Idol," *ACM SIGCSE Bulletin,* vol. 41, no. 3, pp. 4-5, September 2009.

[12] R. E. Pattis, Karel the robot: a gentle introduction to the art of programming, John Wiley & Sons, Inc., 1981.

[13] CodeCombat, Inc., "CodeCombat - Learn how to code by playing a game," CodeCombat, Inc., 2018. [Online]. Available: https://codecombat.com/. [Accessed 11 June 2018].

[14] D. J. Malan and H. H. Leitner, "Scratch for budding computer scientists," in *ACM Sigcse Bulletin*, Covington, 2007.

[15] Google, Inc., "Blockly | Google Developers," Google, Inc., 2018. [Online]. Available: https://developers.google.com/blockly/. [Accessed 5 June 2018].

[16] Google, Inc., "Made with Code | Google," 2018. [Online]. Available: https://www.madewithcode.com/. [Accessed 8 June 2018].

[17] The Walt Disney Company, "Hour of Code | Disney Partners," The Walt Disney Company, 2018. [Online]. Available: http://partners.disney.com/hour-of-code. [Accessed 8 June 2018].

[18] UC Berkeley Computer Science Division, "CS61A Spring 2018," 2018. [Online]. Available: https://inst.eecs.berkeley.edu/~cs61a/sp18/articles/about.html. [Accessed 8 June 2018].

[19] M. de Raadt, M. Toleman and R. Watson, "Training strategic problem solvers," *ACM SIGCSE Bulletin,* vol. 36, no. 2, pp. 48-51, 2004.

[20] Simon, "Soloway's Rainfall Problem Has Become Harder," in LATICE '13 Proceedings of the 2013 Learning and Teaching in Computing and Engineering, 2013.

[21] E. Soloway, "Learning to program= learning to construct mechanisms and explanations," *Communications of the ACM,* vol. 29, no. 9, pp. 850-858, 1986.

[22] F. E. V. Castro, S. Krishnamurthi and K. Fisler, "The impact of a single lecture on program plans in first-year CS," in *Proceedings of the 17th Koli Calling Conference on Computing Education Research*, 2017.

[23] M. Felleisen, R. B. Findler, S. Krishnamurthi and M. Flatt, How to Design Programs, Second Edition, MIT Press, 2014.

[24] S. Bloch, J. Clements, M. Felleisen, R. Findler, K. Fisler, M. Flatt, V. Proulx and S. Krishnamurthi, "Program by Design," [Online]. Available: http://www.programbydesign.org/who. [Accessed 20 May 2018].

[25] E. Schanzer, K. Fisler, S. Krishnamurthi, E. Youndtsmith and R. Sobota, "Bootstrap :: Hour of Code," 2018. [Online]. Available: http://www.bootstrapworld.org/materials/spring2018/courses/hour-of-code/#. [Accessed 12 June 2018].

[26] E. Schanzer, E. Youndtsmith, K. Fisler, S. Krishnamurthi and J. Politz, "Unit 4: The Design Recipe," Bootstrap, 2017. [Online]. Available: http://www.bootstrapworld.org/materials/spring2017/courses/bs1/units/unit4/index.html. [Accessed 20 May 2018].

[27] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information.," *Psychological review,* vol. 63, no. 2, p. 81, 1956.

[28] K. D. Bromley, "Précis writing and outlining enhance content learning," *The Reading Teacher,* vol. 38, no. 4, pp. 406-411, 1985.

[29] D. L. Schacter, The seven sins of memory: How the mind forgets and remembers, Houghton Mifflin Harcourt, 2002.

[30] S. McIntosh, Y. Kamei, B. Adams and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A

case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, Hyderabad, 2014.

[31] Y. a. L. H. Wang, Y. Feng, Y. Jiang and Y. Liu, "Assessment of programming language learning based on peer code review model: Implementation and experience report," *Computers & Education,* vol. 59, no. 2, pp. 412-422, 2012.

[32] T. Busjahn and C. Schulte, "The use of code reading in teaching programming," in *Proceedings of the 13th Koli Calling international conference on computing education research*, Koli, 2013.

[33] GitHub, "Features · Code review," GitHub, [Online]. Available: https://github.com/features/code-review. [Accessed 27 May 2018].

[34] Atlassian, "Crucible Code Review Tool for Git, SVN, Perforce and More," Atlassian, 2018. [Online]. Available: https://www.atlassian.com/software/crucible. [Accessed 27 May 2018].

[35] JetBrains, "Upsource: Code Review, Project Analytics, and Team Collaboration by JetBrains," JetBrains, [Online]. Available: https://www.jetbrains.com/upsource/. [Accessed 27 May 2018].

[36] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *Software Engineering, 2003. Proceedings. 25th International Conference on Software Engineering*, Portland, 2003.

[37] D. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?," *IEEE Software,* vol. 25, no. 2, 2008.

[38] C. Desai, D. Janzen and K. Savage, "A survey of evidence for test-driven development in academia," *ACM SIGCSE Bulletin,* vol. 40, no. 2, pp. 97-101, 2008.

[39] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," *ACM SIGCSE Bulletin,* vol. 36, no. 1, pp. 26-30, 2004.

[40] K. Beck, "Aim, fire [test-first coding]," *IEEE Software,* vol. 18, no. 5, pp. 87-89, 2001.

[41] Z. J. Wood, J. Clements, Z. Peterson, D. Janzen, H. Smith, M. Haungs, J. Workman, J. Bellardo and B. DeBruhl, "Mixed Approaches to CS0: Exploring Topic and Pedagogy Variance After Six Years of CS0," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, Baltimore, 2018.

[42] California Polytechnic State University, San Luis Obispo, "Computer Engineering (CPE) < California Polytechnic State University," California Polytechnic State University, San Luis Obispo, 2018. [Online]. Available: http://catalog.calpoly.edu/coursesaz/cpe/. [Accessed 28 May 2018].

[43] R. Coe, "It's the Effect Size, Stupid," in Paper presented at the British Educational Research Association annual conference, 2002.

[44] J. Cohen, Statistical power analysis for the behavioral sciences. Revised Edition, New York: Academic Press, 1977.

[45] G. V. Glass, M. L. Smith and B. McGaw, Meta-analysis in social research, Sage Publications, Incorporated, 1981.

[46] Google, Inc., "Cloud Natural Language | Google Cloud," Google, Inc., 2018. [Online]. Available: https://cloud.google.com/natural-language/. [Accessed 5 June 2018].

[47] Google, Inc., "Natural Language API Basics | Cloud Natural Language API Documentation | Google Cloud," Google, Inc., 2018. [Online]. Available: https://cloud.google.com/natural-language/docs/basics#sentiment-analysis-values. [Accessed 11 June 2018].

[48] T. Vanderbilt, "Let the robot drive: The autonomous car of the future is here," *Wired Magazine, Conde NAST, www. wired. com,* pp. 1-34, 2012.

[49] Summer of Innovation Zero Robotics, "Making a Peanut Butter and Jelly Sandwich," [Online]. Available: http://static.zerorobotics.mit.edu/docs/team-activities/ProgrammingPeanutButterAndJelly.pdf. [Accessed 20 May 2018].

[50] Microsoft Corporation, "T.TEST function - Office Support," Microsoft Corporation, 2018. [Online]. Available: https://support.office.com/en-ie/article/t-test-function-d4e08ec3-c545-485f-962e-276f7cbed055. [Accessed 5 June 2018].

APPENDICES

A. Architecture Design of DRCOP

The architecture of DRCOP contains two main subsystems and a single abstract data type definition that serves as the medium of information flow between the two subsystems. An abridged UML diagram illustrating this architecture is presented below in Figure A.1.

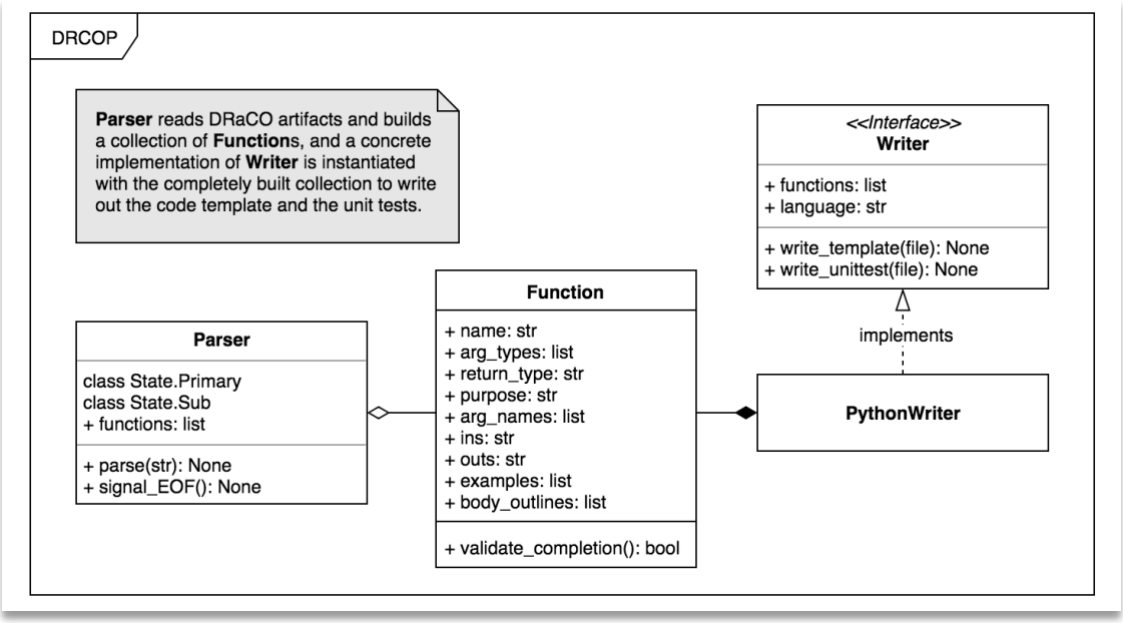


Figure A.1 Abridged UML of the high-level architecture of DRCOP

Main Components of DRCOP

The abstract data type, class **Function**, represents a function in a program, with the specifications of a function such as name, input types, return type, purpose, argument names, side-effects, examples, and the code outline for the function body as the attributes.

First of the subsystems is the **Parser** class, which is a state machine responsible for reading the design recipe blocks and building a collection of functions. It also validates that the specifications for a function expressed in the design recipe is complete with all of its components required to be used in a function stub generation. The parser reads and processes one line of the `*.oln.py` at a time and uses the header of the design recipe lines, e.g. `CONTRACT`, `PURPOSE`, etc., and other textual artifacts to determine which state it needs to be in to properly tokenize and populate the collection of functions with the line of text it is processing. This is a theoretically reliable method of determining the state. Despite the fact that input text is arbitrary, the electronic design recipe template we provide to the students ensures that virtually no student would hand-write the header of the design recipe lines. The states the parser maintains are composed of 'primary' ans 'sub' states implemented as inner classes within parser. The primary state is designed to distinguish the design recipe block from the function-level outlines, and the sub state is kept track to separately track each line of the design recipe block.

The second subsystem is the **Writer** class, which is designed at a high-level as a language-independent interface requiring the concrete implementation of template and unit test composition strategies to provide any language-specific details. Since CPE-101 only uses Python, we construct the concrete **PythonWriter** class that implements writer to generate the Python syntax. Our implementation of the Python writer is quite simple. Once the parser completes processing the `*.oln.py` file and the driver of DRCOP instantiates the writer with the collection

of function and the file I/O details, our writer simply assembles the substrings of Python unit test syntax corresponding to the design recipe's components and writes them sequentially to the two output files.

These subsystems and the abstract data type are put together as a single pipeline inside of a command-line utility, written in Python and wrapped around a BASH shell script for the proper delivery of the error logs to the instructor of the course. This dual-layer driver is also responsible for input validation, file I/O, error handling, logging, and helping students avoid small operational mistakes such as file overwrites that may result in data loss.

Error Handling During Parsing

We designed DRCOP as a tool that students can use freely as they are working on the course projects. This implies that the parser is exposed directly to the arbitrary input fed in by the novice students, with even the best-case scenario including some inevitable typos or mistakes in the design recipe that students write. Thus, we include three different levels of error handling in the design of DRCOP to flexibly handle various error scenarios while attempting to minimize student frustration in dealing with the tool.

First and the least sever level of error case is '`Ignorable PARSE ERROR`.' We recognize that our target audience is students who are not proficient at reading and comprehending uncaught exception messages and stacktrace. Therefore, we minimize the need to for the human user to handle any errors that occur during the execution of DRCOP if the error can be somehow reconciled by the tool. When some benign error is found in the text being parsed, DRCOP reports any such error,

while still ‘doing its best’ in a *JavaScript*-like attitude to produce the output files that are written in valid Python syntax. Certainly, this presents some possibility of embedding unpredictable behaviors or hard-to-catch bugs in the code generated. Nevertheless, this is a tradeoff we purposefully permit to prevent students from ever getting completely ‘blocked’ from proceeding to run the generated unit tests if they wish.

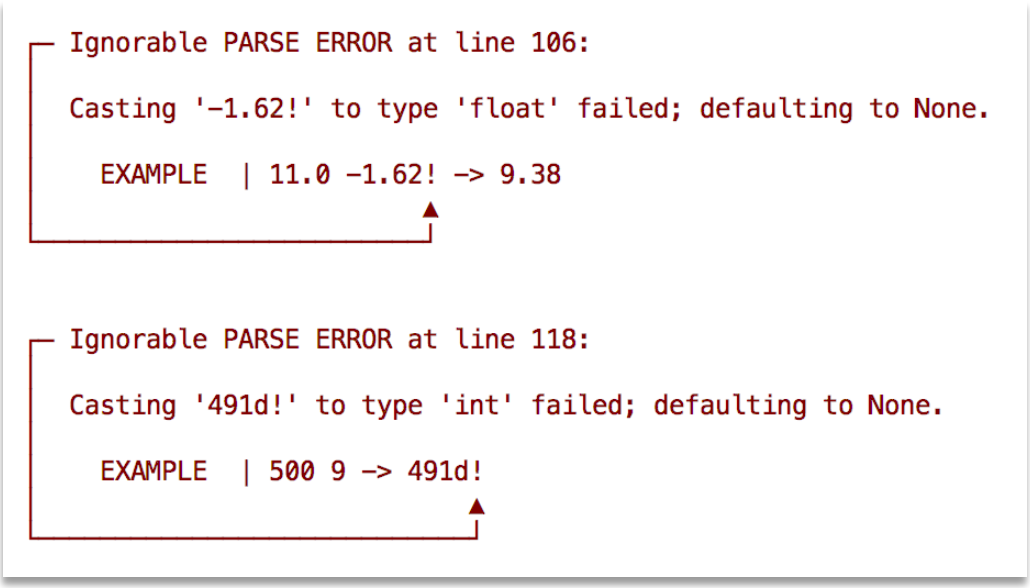


Figure A.2: Parse errors from DRCOP being presented as ‘Ignorable’

Figure A.2 above, which is a screen capture from the standard error stream, illustrates this. The messages shown here are a result of unexpected characters in the `EXAMPLE` line. DRCOP expects to fine a floating-point number, but some mistake from the student results in pollutive characters to cause typecast failures when generating unit tests. The tool reports these as `Ignorable PARSE ERROR`s and continues to run, eventually generating the output files.

Although `Ignorable PARSE ERROR`s of DRCOP are quite similar to “warning” messages of many conventional programming languages’ compilers or

interpreters, we decide to stick with the term ‘ignorable’ instead to explicitly communicate that DRCOP’s execution is not interrupted. This again comes from understanding that most novice students do not know the difference between “warnings” and “errors.” This terminology is designed to present a clear choice between the following options to the students: (1) addressing the issues that DRCOP runs into while parsing, or (2) simply accepting the fallback measures that it provides.

The next level of severity in DRCOP’s error hierarchy is ‘`CRITICAL PARSE ERROR`’s. Expected to be encountered much less frequently than the ‘ignorable’ errors, ‘critical’ errors correspond to cases where certain unexpected content from the DRaCO text severely disrupts the operation of the parser. A student’s attempt to run DRCOP on a high-level (instead of function-level) code outline or some other non-DRaCO text can result in this disruption. In such cases, DRCOP reports this error and halts, directing students to double check their outline. Figure A.3 below shows an example of this, caused by a student’s attempt to run DRCOP on a high-level code outline that does not have any design recipe blocks.

```
CRITICAL PARSE ERROR at line 8:

Invalid syntax caused function object to not populate (check your outline).

  ## 1) We need to ask the user for s ...
  ▲
```

Figure A.3: Halting parse errors from DRCOP being reported as ‘CRITICAL’

In both levels of `PARSE ERROR` handling, we carefully design all student-facing error messages to be novice-friendly. Despite the restrictions the command-

line interface places on visual communications, we still determine that short error messages with ASCII-art style graphics can communicate the causes of the errors better than paragraphs-worth text instructions. Thus, whenever applicable, we completely hide any conventional stacktrace-style errors and show a single-sentence description of the error with a long-tailed arrow pointing to the portion of their source text that caused the error instead. The long tail of the arrow also serves as a box-shaped visual separator of multiple error messages in order to reduce any potential confusion caused by a handful of error messages filling the whole output screen.

The last and most severe level of the error is caused by a propagation of an uncaught exception to the driver (main function) of DRCOP during a runtime. Because the errors caused by the unexpected text from the students' input is handled mostly by the `PARSE ERROR`s, this case is most likely caused by some defective internal logic of DRCOP. Therefore, we have implemented a detailed error logging functionality such that the error conditions and the stacktrace from the error is delivered to a UNIX directory designated by the instructor for troubleshooting. If this error case is triggered, DRCOP generates an 'error code' and displays it to the student, along with a message instructing the student to contact the instructor with that code.

Operational Detail

We present the operational details of our tool as a sequence diagram shown below in Figure A.4. It shows the high-level structure of DRCOP's architecture design, and how each component of it interacts over the lifetime of a single execution of

the tool. To provide better context, we present four more components in the diagram in addition to the three key components shown in Figure A.1.

On the left-hand side of the sequence diagram, we show lifelines of each layer of DRCOP's dual-layer: DRCOP Wrapper (written in BASH) and DRCOP Main (written in Python). On the right-hand side of the diagram, we show two system-level components that interact heavily with DRCOP but is not part of the tool: Console I/O (`stdout` and `stderr`), and File I/O.

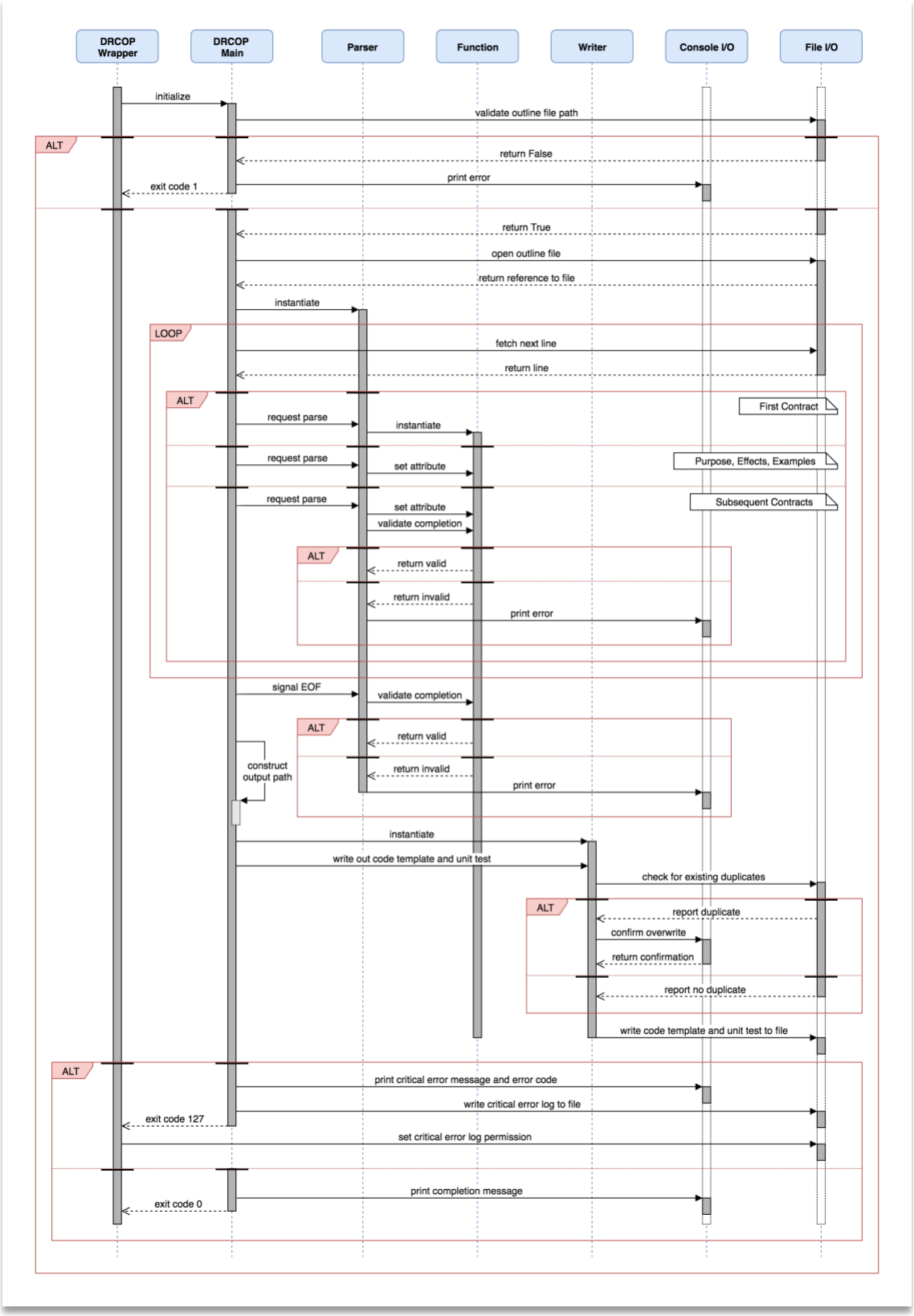


Figure A.4: Detailed sequence diagram of DRCOP's execution

B. Instructions for DRCOP Usage

This document provides a quick guide on how to use DRCOP to generate a function template file and a unit test file for your program development.

1. If you haven't done so already, use *Cyberduck*, *FileZilla*, or `scp` to transfer your code outlines for the functions file (for example, `my_funcs.oln.py`) to the Cal Poly's UNIX machine.
2. Using Terminal or Git Bash, SSH to one of the CSC UNIX machines (e.g. `ssh username@unix3.csc.calpoly.edu`).
3. Navigate to the directory you had transferred your files to (most likely `~/cpe101/labX/ or ~/cpe101/projectX/`).
4. Check the directory listing with ls command to make sure your function file (for example, my_funcs.oln.py) is in the current directory.
5. Type in the following command to convert your code outline file to a code template and generate the unit tests. Be sure to replace '`my_code_outline`' with the actual file name of your outline:

   ```
   //home/doryu/services/DRCOP my_funcs.oln.py
   ```

6. You may see "PARSE ERROR" that show up as you run DRCOP. Although many of them are marked as "ignorable," you might want to go back and check your outline to make sure if you haven't made any mistakes, as any PARSE ERROR generally leads to some unexpected or incorrect generation of the function template and the unit tests. Repeat Step 5 and 6 as necessary.

7. Try the `ls` command again, and you'll notice that the unit test file (e.g. `my_funcs_test.py`), as well as the template file (e.g. `my_program.py`) has been generated.
8. Try running the generated unit test file. Most of the test should fail, because your code hasn't been written yet.
9. At this point, you're ready to write actual code. Open up or transfer the template generated (e.g. `my_funcs.py`) and begin writing code as outlined. As you complete your functions, try-re-running the unit test and check if more tests are passing. You'll know you've successfully implemented your functions when you see all tests pass! (Assuming that your examples were written correctly ...).

C. Selection of Student-Composed Linear Code Outlines

The code outline shown in Figure 4.2.4: An example code outline for a simple program and the outlines reproduced here are for the same ‘skater’ project.

Sample Linear Code Outline 1

```
#first ask how much the user weighs
#convert answer to float
#convert weight of skater to KG with poundstoKG
#then ask how far away the professor is

#convert distance to float
#then ask what type of object they want to throw
#use getMassObject function to find the mass of the object
#use getVelocityObject to find the velocity of the object
#Using the mass of the skater, mass of object, and velocity
of object calculate velocity of skater with getVelocitySkater
#velocity should be reported
#using if statements report appropriate remark based on
velocity of object
```

Sample Linear Code Outline 2

```
#first, we will need to ask our user how much they weigh in pounds.
#we will do this through using input to allow them to return an answer
#second, we will ask them how far away the professor is (in meters).
#third, we will ask them what item they will choose to throw.
#depending on the weight of the object (in kilograms) that the user chooses to throw, the print statement we choose to show will vary.
# we will print the correct statements by using if, elif, else regarding the mass of the object
#after this, we will determine the velocity of the skater by importing the function we created in funcs.oln.py - then we will print the velocity and a corresponding statement.
#after this, we will again use if, elif, and else to determine which statement to print that corresponds with the given skater velocity, per the instructions.
```

Sample Linear Code Outline 3

```
# ask user for their weight in pounds
```

```
# ask user for distance (in meters) between skater and
professor
# ask user which object they would like to throw
# calculate the velocity of the object
# calculate the velocity of the skater and print the result
```

Sample Linear Code Outline 4

```
# Import functions from the func.py file
# Ask the user for their weight (lbs) and cast the input as
a float
# convert weight (lbs) into mass (kg)
# Ask for user distance to professor
# calculate velocity of the object
# Ask user to input the object they are throwing
# calculate mass of object
# calculate velocity of skater
# print based on the variables in previous steps.
```

D. Project Specifications from CPE-101

Below, we reproduce the DRaCO workflow instructions from the specifications of Projects 1 and 4 published to our experimental group. More project specifications with different instructions for each project used in CPE-101 is available on the online extension of this appendix at: http://mikeryu.com/DRaCO.

"Code Outline and Peer Review Process" from Project 1

As we discussed in class, we will utilize the code outlining process before we start writing the code. For this project, you are required to submit two code outlines; one for your functions (`funcs.oln.py`), and the other for your main program (`skater.oln.py`).

Follow the steps below to complete this phase of the project:

1. Do a close read of this specifications, top-to-bottom. As you're reading the specifications, think about what steps are necessary to generate the required output.

2. If you haven't done so already, set up your **Sublime Text** to support a design recipe snippet. Visit http://mikeryu.com/dr to learn how to set this up.

3. Start on your first outline, `funcs.oln.py`. This file will serve as the outline for your functions file, namely `funcs.py`. For this particular project, design recipe will contain most of the important information, with the outline for function body fairly minimal.

   An example of how your `funcs.oln.py` should start is shown below (some information is obfuscated with '...', but your design recipe and outline should be complete):

```
"""
Project 1

Name: Boaty MacBoatface
Instructor: Mike Ryu
```

```
Section: 13
"""

"""
CONTRACT | poundsToKG : ...
-------: | :-----------------------------------------------
---
PURPOSE  | Converts weight in `pounds` ...
EFFECTS  | ...
EXAMPLE  | 0.0 -> 0.0 # 0 pound is 0 kg
         | 1.0 -> ...
         | ...
"""
# calculate the result with given formula
```

4. Once you've completed `funcs.oln.py`, start your outline for your main program in `skater.oln.py`. This file will consist of the file header (see the subsequent section), as well as a few in-line comments that list the steps necessary to complete the main program.

   An example outline is shown below (note that this example is intentionally written tersely as to not spoil any fun for you -- your outline should be much more detailed):

```
"""
Project 1

Name: Boaty MacBoatface
Instructor: Mike Ryu
Section: 13
"""

# first need to ask user about something

# then another thing

# now that we have some data, do some calculations

# report the result!
```

5. As soon as you're done with `funcs.oln.py` and `skater.oln.py`, you may handin the two outlines using the following command:

   ```
   handin grader-ph 101project01 funcs.oln.py skater.oln.py
   ```

   Code Outline is due on Saturday, 1/20 by 11:59 pm via handin

6. Once you have handed in your code outlines, you may convert your `funcs.oln.py` to the template and corresponding unit test files using the process outlined in http://mikeryu.com/oln. At this point, you should begin writing actual code.

7. We will do an in-lab exercise on Monday, 1/22 to peer review each other's code outline, to make sure that everyone's thought process expressed in the outline is well organized and logically sound.

"Design Recipe and Code Outlines (DRaCO)" from Project 4

Due *THIS* Friday, 2/23 by 10:00 am via handin

The process of carefully designing your functions and expressing the implementation plan in terms of code outlines ***has never been more important!***

Therefore, the deliverable for this portion will compose
whopping **40% of your Project 4 grade**.

Please read the following requirements ***carefully*** and handin your deliverables ***on time***:

1. Read the subsequent pages of this specifications to understand the problem.

2. Think carefully about which functions you should write. Consider the following:
   - Are your functions easy to understand? What would be their `PURPOSE`?
   - Are your functions easy to test? Are you able to think of a few `EXAMPLE`s?

3. Once you've decided on which functions to write …
   - Write the DRaCO for your functions in `funcs.oln.py` file.
     - Each function's DR must have at least ***FIVE* (*5*)** distinct `EXAMPLE`s.
     - Each function's DR must be followed by a detailed CO. (***No code, tho!***)
   - Generate the unit tests and the template file using DRCOP.

        - Your DRaCO must not cause ***any*** ERRORs when being processed.

4. Once you've completed step 3 …
    - Design your `word_finder.py` by writing `word_finder.oln.py`.
        - Only write the detailed CO for the main program (no Python code).
        - Think about how your functions will be used, in what order.
        - Clearly indicate how you'll be using conditionals, loops, etc.
    - `word_finder.oln.py` does ***not*** need to be processed by DRCOP.

5. Handin your work, both DRaCO ***and*** the generated Python unit test and template files.
    - First, navigate to the UNIX folder where you've uploaded your work.
    - Then, use this command:

```
handin grader-ph 101project04_ryu *.py
```

6. On Friday, 2/23, we will perform an in-depth code review of your DRaCO.

If you do not have the submission ready by 10:00 am, you'll receive 0% credit.

E. In-Depth Code Review Worksheets

In-Depth DRaCO Review Sheet for Project 4

In-depth DRaCO Review
CPE 101-14, Winter 2018

This review will determine the DRaCO *portion grade for the reviewer and the author!* **Project 4** Word Finder

REVIEWER NAME		**AUTHOR NAME**	

Carefully review the DRaCO assigned to you and answer the following questions. You might want to speak with the author to ask some questions if you require some assistance to understand the intention of the author.

1. How many I/O function(s) does the DRaCO in `funcs.oln.py` have? What are their names?

2. Review the functions from this DRaCO in `funcs.oln.py` that do ***not*** do I/O. Which functions(s) have the responsibility to perform the following tasks? Write the corresponding function name(s) underneath each task.

a. Finding the given word in the puzzle that appears in "forward" direction.

b. Finding the given word in the puzzle that appears in "up" direction.

c. Finding the given word in the puzzle that appears in "diagonal" direction.

3. Carefully review the `EXAMPLES` from the function you named in **2c**. If the author followed the project specifications, each of the `EXAMPLE` should test distinct scenarios. Describe what each of those scenarios are. *Feel free to ask the author of the* DRaCO *for this.*

a. _____
b. _____
c. _____
d. _____
e. _____

4. Now, shift your focus to `word_finder.oln.py`. Does the code outline clearly indicate which of the functions from `funcs.oln.py` will be used where and how? Write below, ***in order***, the names of functions that will be called in the DRaCO author's `word_finder.py`.

5. Just by looking at the DRaCO, can you determine whether their design can handle finding palindromic words such as `MADAM`? If so, describe how you think such cases will be handled based on the DRaCO. *If not, ask the original author and summarize their response*.

6. What was one design decision this author has made that was similar to your own DRaCO? Why do you think that particular decision is important for this project?

7. What was one design decision this author has made that was different from your DRaCO? Pick one that you thought was most ingenious or confusing and explain what the difference is, and why you thought it was ingenious or confusing.

8. Make ***one*** suggestion to improve the author's program design. Even if there aren't any blatantly obvious logical errors or mistakes, think about whether there is any room for code reuse, or if there are any corner cases that should've been written as `EXAMPLES`.

9. Rate the overall quality of this author's work as a letter grade (***circle one***).

A (Excellent) **B** (Good) **C** (Average) **D** (Needs Work) **F** (Poor)

In-Depth DRaCO Review Sheet for Project 6.1

In-depth DRaCO Review

CPE 101-14, Winter 2018

This review will determine the DRaCO portion grade for the reviewer and the author! **Project 6 Part I** Pixel Magic

REVIEWER NAME		**AUTHOR NAME**	

Carefully review the DRaCO assigned to you and answer the following questions. You might want to speak with the author to ask some questions if you require some assistance to understand the intention of the author.

1. Read through the outlines written by the author in `encode.oln.py` and `decode.oln.py` and try your best to identify the following. Circle **Y**es, **N**o, or **C**an't tell. Choosing "No" or "Can't tell" doesn't necessarily imply anything negative about the author's DRaCO.

   a. Does the outline show how the author is planning on opening the file(s)? **Y** / **N** / **C**

   b. Does the outline show the plan for encoding and decoding pixels? **Y** / **N** / **C**

   c. Does the author process and write out pixels one at a time, or do they process the entire image in a data structure, then write the whole image to the file? **Y** / **N** / **C**

   d. Is there any information about error handling (*e.g.* while handling the file I/O) in `encode.oln.py` and `decode.oln.py`? **Y** / **N** / **C**

2. Now, turn your attention to the author's `pixel_io.py` and answer the following questions.

   a. Reproduce the `CONTRACT` of the function which opens the original image file.

   b. Reproduce the `CONTRACT` of the function which reads from the original image file.

   c. Reproduce the `CONTRACT` of the function that writes a pixel to the output file.

   d. Reproduce one `CONTRACT` of the function that outputs some information to the screen. If there isn't one, do you think there should be one (or more)?

   e. If your response to **1d** was not a "Yes" (if it was, circle **NA**) do you see the error handling logic written in any of the functions in `pixel_io.py`? **Y** / **N** / **NA**

3. Now, shift your focus to `pixel_lib.oln.py` and answer the following questions.

a. Scan the `CONTRACT` and `PURPOSE` of the functions. Are there any function names that are ambiguous or do not seem to correctly represent their purpose? If there were any, list the names of those functions below.

b. Are there any functions that use other functions within `pixel_lib.oln.py`? If there are any, list the names of those functions below.

c. Did the author write useful and distinct `EXAMPLE`s, at least 3 per function? **Y** / **N**

d. Did the author write clear and concise `PURPOSE` statements for all functions? **Y** / **N**

e. Did the author sufficiently outline the implementation plan for every function? **Y** / **N**

4. What was one design decision this author has made that was similar to your own DRaCO? Why do you think that particular decision is important for this project?

5. What was one design decision this author has made that was different from your DRaCO? Pick one that you thought was most ingenious or confusing and explain why.

6. Make ***one*** suggestion to improve the author's program design. Even if there aren't any blatantly obvious logical errors or mistakes, think about whether there is any room for code reuse, or if there are any corner cases that should've been written as `EXAMPLES`.

7. Rate the overall quality of this author's work as a letter grade (***circle one***).

A (Excellent) **B** (Good) **C** (Average) **D** (Needs Work) **F** (Poor)

PDF versions of these documents are available on the online extension of this appendix at http://mikeryu.com/DRaCO.

F. Midterm and Lab Final Exam Problems from CPE-101

The following exam documents from CPE-101 are available on the online extension of this appendix at http://mikeryu.com/DRaCO as PDF files:

- Midterm I, Part 2 (Problems 6 through 9)
- Midterm II, Part 2 (Problems 8 and 9)
- Lab Final Exam Specifications
- Supplemental Source Code for Lab Final Exam

G. Student Exit Interview Results

The entirety of student exit interview results is available on the online extension of this appendix at http://mikeryu.com/DRaCO in the following formats:

- Spreadsheet with Anonymized Responses
- Distribution Plots for Each Exit Interview Question

H. Source Code of DRCOP

Source code of DRCOP is freely available for under the GNU Lesser General Public License, version 3 (LGPL-3.0). You can download the ZIP archive of the source code from the online extension of this appendix at http://mikeryu.com/DRaCO. Public source code repository of DRCOP is also available on GitHub: https://github.com/mikeryu/ms-thesis.