DePaul University

Digital Commons@DePaul

College of Computing and Digital Media Dissertations

Jarvis College of Computing and Digital Media

Spring 4-21-2023

Code generation based on inference and controlled natural language input

Howard R. Dittmer
DePaul University, hdittmer@mac.com

Follow this and additional works at: https://via.library.depaul.edu/cdm_etd

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Recommended Citation
Dittmer, Howard R., "Code generation based on inference and controlled natural language input" (2023). *College of Computing and Digital Media Dissertations*. 46.
https://via.library.depaul.edu/cdm_etd/46

This Dissertation is brought to you for free and open access by the Jarvis College of Computing and Digital Media at Digital Commons@DePaul. It has been accepted for inclusion in College of Computing and Digital Media Dissertations by an authorized administrator of Digital Commons@DePaul. For more information, please contact digitalservices@depaul.edu.

CODE GENERATION BASED ON INFERENCE AND CONTROLLED NATURAL LANGUAGE INPUT

BY

HOWARD R. DITTMER

A DISSERTATION SUBMITTED TO THE JARVIS COLLEGE OF COMPUTING
AND DIGITAL MEDIA OF DEPAUL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF
DOCTOR OF PHILOSOPHY

DEPAUL UNIVERSITY
CHICAGO, ILLINOIS
2023

DePaul University

Jarvis College of Computing and Digital Media

Ph.D Computer and Information Sciences Dissertation Defense

This doctoral dissertation has been read and approved by the dissertation committee below according to the requirements of the Computer and Information Systems PhD program and DePaul University.

Name:
HOWARD R. DITTMER

Title of dissertation:
CODE GENERATION BASED ON INFERENCE AND CONTROLLED NATURAL LANGUAGE INPUT

Date of Dissertation Defense:
APRIL 21, 2023

Dissertation Committee:
XIAOPING JIA, Committee Chair / Dissertation Advisor*
ADAM STEELE, Committee Member / 1st Reader
WAEL KESSENTINI, Committee Member / 2nd Reader
CHRIS JONES, Committee Member / 3rd Reader

** A copy of this form has been signed, but may only be viewed after submission and approval of FERPA request letter.*

CODE GENERATION BASED ON INFERENCE AND CONTROLLED NATURAL LANGUAGE INPUT

Abstract

Over time the level of abstraction embodied in programming languages has continued to grow. Paradoxically, most programming languages still require programmers to conform to the language's rigid constructs. These constructs have been implemented in the name of efficiency for the computer. However, the continual increase in computing power allows us to consider techniques not so limited. To this end, we have created **CABERNET**, a Controlled Natural Language (CNL) based approach to program creation. **CABERNET** allows programmers to use a simple outline-based syntax. This syntax enables increased programmer efficiency.

CNLs have previously been used to document requirements. We have taken this approach beyond the typical application of creating requirements documents to creating functional programs. Using heuristics and inference to analyze and determine the programmer's intent, the **CABERNET** toolchain can create functional mobile applications. This approach allows programs to align with how humans think rather than how computers process information. Using customizable templates, a **CABERNET** application can be processed to run on multiple run-time environments. Since processing a **CABERNET** program file results in a native application program, performance is maintained.

This research explores whether a CNL-based programming tool can provide a readable, flexible, extensible, and easy-to-learn development methodology. To answer this question, we compared sample applications created in Swift, SwiftUI, and a prototype of the **CABERNET** toolchain. The **CABERNET** implementations were consistently shorter than those produced in the other two languages. In addition, users surveyed consistently found the **CABERNET** samples easier to understand.

Acknowledgements

This degree has been a long journey for me. Across that time there have been a number of people whose support has been important.

First I must thank Dr. Xiaoping Jia. He has been my advisor, mentor and support throughout this process. Without that support, I would likely have not made it to this point. Dr. Jia has been my advisor since I started my Master's Degree twenty-plus years ago. I am not sure either of us thought we would get to this outcome. I would also like to thank the other members of my dissertation committee: Chris Jones, Adam Steele and Wael Kessentini for their support in this process. Chris in particular has shared Dr. Jia as his advisor and has been very supportive of me over the years.

My wife, the love of my life and best friend for forty-eight years and counting, Sheree, has been my rock. This work would not have been possible without her support, understanding and encouragement. All the times I disappeared into my office and zoned into this work would not have been possible without her understanding and support.

Add to that the encouragement I have received from my mom, our two sons, my friends and co-workers over the years have made this possible. I wish my mom was still here to see this.

Contents

List of Tables

List of Figures

List of Abbreviations

AI	**A**rtificial **I**ntelligence
API	**A**pplication **P**rogramming **I**nterface
AST	**A**bstract **S**yntax **T**ree
BDD	**B**ehavior-**D**riven **D**evelopment
CABERNET	**C**ode gener**A**tion **B**as**E**d on cont**R**olled **N**atural languag**E** inpu**T**
CASE	**C**omputer **A**ided **S**oftware **E**ngineering
CNL	**C**ontrolled **N**atural **L**anguage
GUI	**G**raphical **U**ser **I**nterface
IA	**I**ntelligence **A**ugmentation
IDE	**I**ntegrated **D**evelopment **E**nvironment
ISDOS	**I**nformation **S**ystem **D**esign and **O**ptimization **S**ystem
OWL	**W**eb **O**ntology **L**anguage
UML	**U**niveral **M**odeling **L**anguage

Chapter 1

Introduction

Computer programming is about providing tools for improving the productivity of human users. The tools that are embodied in computer programs have improved the productivity of all types of users. However, one area that can still benefit from computer-based automation is program development. While many tools automate specific tasks performed by a programmer, there is a lack of consistent automation directed at the actual process of creating instructions that embody the program.

Computer-aided Software Engineering (CASE) tools have existed since the late 1960s. In 1968 researchers at the University of Michigan started the Information System Design and Optimization System (ISDOS) project [1]. This project had the goal of developing a "...problem statement technique." The ISDOS project was the beginning of requirements management as a CASE tool. In 1973 Terry Winograd argued for an "intelligent assistant"[**memo:leIpigfi**] to handle many of a programmer's routine tasks. These are but two alternate visions for tools to aid program developers.

In Figure 1.1, we have captured the range of techniques involved in computer-assisted programming. These approaches include everything from requirements capture and code generation to evaluation of code for potential errors. We will discuss the nature of the broad base of research on these approaches in this paper's

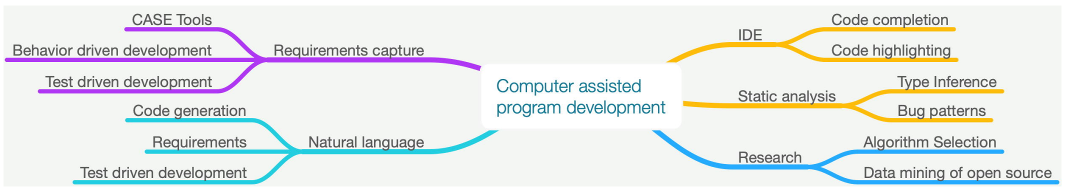


FIGURE 1.1: Techniques for computer-assisted programming.

background chapter. Unfortunately, most of these tools have been applied piecemeal to the program development domain. By starting with code generation based on inference and a controlled natural language, we see an opportunity to address the programmer's core function, actual code generation.

Much has been made of using Artificial Intelligence (AI) to replace human efforts. In the field of program development, there is an expectation that AI could generate computer programs based on the input of requirements. Some recent work has involved the use of Machine Learning to generate code. This effort seeks to replace the human programmer's efforts. Alternately, we see our efforts as not an attempt to replace the developer but an opportunity to increase the developer's productivity. Our efforts follow the path of Intelligence Augmentation (IA) proposed by Doug Engelbart and Terry Winograd [**fisher:2017vd**, 2, 3]. To that end, this approach combines inference with developer interaction to create robust solutions to program needs while maximizing developer productivity.

While significant research (discussed in Chapter 2) has proceeded us across the range of computer-assisted program development, there still needs to be more progress on actual code generation. The challenge is to create a flexible, intuitive, and natural methodology for the developer. The tool should allow for synonyms, acronyms, abbreviations, and shorthand. It should allow flexibility in the structure of the information provided to it. Most importantly, it must deal with ambiguous and unrecognized content cleanly. Finally, the process must produce unambiguous and consistent results. Our approach meets all these requirements.

Our goals are two-fold. We seek to provide novice developers with a tool and approach that allows them to be productive without the learning curve in existing programming approaches. At the same time, we seek to provide experienced developers with a methodology that improves their productivity.

To this end, we have created a programming methodology named *CABERNET*. As part of our research, we have developed a tool to generate mobile applications based on this methodology. This paper will describe the methodology and example applications built with it. We believe that our approach provides a development platform that can produce deterministic results while allowing flexibility in the input and code. At the same time, this approach is easy to understand and accessible for novices. This combination will provide the opportunity for significant improvements in programmer productivity and quality.

Through this research, we have sought to answer the question: *Can a controlled natural language-based programming tool provide a highly readable, flexible, extensible, and easy-to-learn development methodology?*

Chapter 2 of this paper briefly discusses other approaches to providing tools that address the broad range of program development processes and compare them with our work. Chapter 3 provides a vision for what is possible with this CNL-based programming approach. Chapter 4 describes the core concepts of *CABERNET*. Chapter 5 describes the syntax and structure of a *CABERNET* program. Chapter 6 documents the results of the comparison of *CABERNET* with the alternate approaches. Chapter 7 reviews the current state of this research. Finally, Chapter 8 discusses other approaches to improving programmer effectiveness.

Chapter 2

Background

The programming community continually looks for ways to improve the efficacy of those involved in program development. We define the measure of improvement in programmer efficacy or productivity as a reduction in the quantity of work required to produce a defect-free program or program function. Researchers have long sought to automate various aspects of the software development process. Today there are many tools and techniques available to help developers in their work. Some of these are discussed below. As we will see, few of these directly address code generation.

2.1 Machine Learning

A recent area of activity is computer-assisted programming through machine learning. The approach involves training a tool with libraries or code repositories. Using this resource, the tool then provides code recommendations to the programmer. Examples of this include *Natural Language-Guided Programming* [4], *OpenAI Codex* [5] and *GitHub CoPilot* [6], an OpenAI based code generation tool. This is based on the same technique applied by ChatGPT. ChatGPT can generate some types of program files but not the iOS programs we are considering in this work. Since *GitHub*

CoPilot is based on the same OpenAI work and is applied to programming our analysis will be based on this tool. Another approach in this area is *Genetic programming* [7]. *Genetic programming* is similar in usage to the other two solutions but is based on hand-coded training cases making it much more expensive to implement.

These tools attempt to improve programmers' productivity by providing coded solutions to portions of programs as the developer works. This appears to be a benefit to a programmer, particularly when working with a language or in a solution space with which they are not familiar. The approach that these tools take has been described as AI Pairs Programming or as an automatic code completion tool. Since *GitHub CoPilot* generates code based on examples collected from publicly available code on GitHub there has been some question about the quality of the result. There is no assurance that the source code is correct or efficient. A recent study [8] of the results of *GitHub CoPilot* generated code gives reason for concern. In this study, they found that in 28.7% of the problems *GitHub CoPilot* generated the correct code. In 20.1% of the problems *GitHub CoPilot* completely failed to provide a correct solution. If you combine the 51.2% of the time where the solution is partially correct with the totally correct solutions, you get 79.9% of the time where a solution would be helpful to a programmer. Another study of *GitHub Copilot* generated code [9] found code correctness ranged from 57% on Java examples to 27% on JavaScript examples. But these success rates are not such that a programmer can expect a correct solution without significant review and refinement.

We have applied *GitHub CoPilot* to some of the sample problems we have considered in our work. First, we asked *GitHub CoPilot* to: *"create an iOS app in SwiftUI which accepts a Bill Amount, a Tip Percentage and the number of people paying and calculates the amount each person pays"*. *GitHub CoPilot* offered seventeen solutions to this request. Fifteen of these solutions ran without error. All the proposed solutions matched Figure 2.1. However, none of these solutions looked like the output of our programs (see Figure 5.5b). The *GitHub CoPilot* produced applications that required the user to select their tip amount from a list of possible values. These apps also require the user to select the number of people from a list. Some of these solutions return a total bill amount in addition to the amount each person pays which is different from the instructions and different from our solution. Why would all of these proposed solutions be different from what was requested and different for our solution? It turns out that there is a website titled *100 Days of SwiftUI*[10] that contains a video training course that produces a tip calculator. All of these proposed solutions come from students taking this course and saving their work on GitHub. The problem is that all of these solutions ignored the original instructions we provided to *GitHub CoPilot*. Next, we tried providing more detail in our instructions to *GitHub CoPilot*. We asked *GitHub CoPilot* to *"create an iOS app in SwiftUI which accepts a Bill Amount in dollars, a number between 0 and 35 for the Tip Percentage and a number between 1 and 6 representing the number of people paying and calculates the amount each person pays"*. With these instructions, the solutions provided were the same as the first set of responses. We conclude that *GitHub CoPilot* sees the instructions to create a tip calculator and it provides the examples of tip calculators it finds in the GitHub repository.

To test our conclusion from the tip calculator we offered *GitHub CoPilot* a second example to process. We asked *GitHub CoPilot* to: *"create an iOS app in SwiftUI which accepts the Lot Width and the Lot Depth and calculates the area of the lot by multiplying the Lot Width and Lot Depth dividing by 43560 square feet per acre"*. To this request, *GitHub CoPilot* provided eight potential solutions. Five of these solutions successfully ran and calculated the area of the presented lot. However, each of the solutions included more or different information than what was included in the

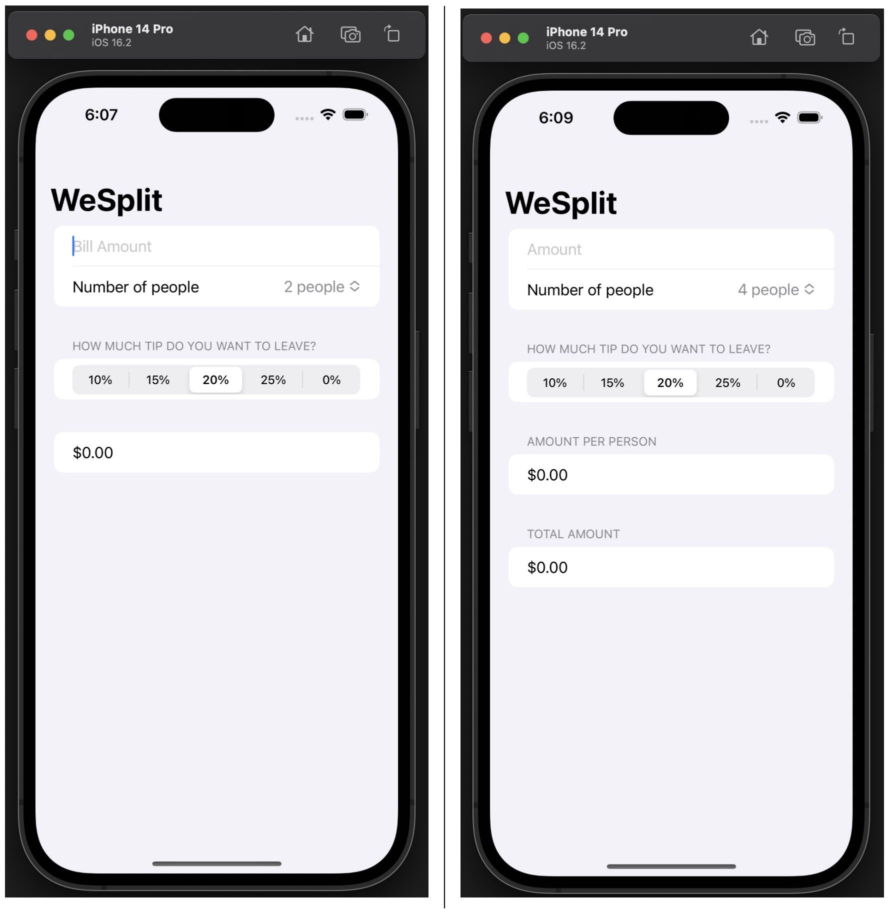


FIGURE 2.1: Tip calculators produced by CoPilot.

instructions and was organized differently. The resulting solutions can be seen in Figure 2.2. So, while this tool has provided program solutions to these relatively well know problems the solutions do not correspond to the specifics of the instructions provided. The result is that the programmer has little or no control over the specifics of the resulting program.

These results indicate that *GitHub Copilot* and the others will likely be valuable tools for programmers in the future[11]. However, given the questionable quality of the code source (GitHub public repository), there will continue to be a need for a close review of the results. Additionally, these are tools to aid programmers in their development efforts, not tools for actually creating programs. As we have seen the solutions provided may match the high-level requirements provided the tools allow the programmer little or no control over the details of the program.

2.2 Controlled Natural Languages

A natural language programming technique has long been a goal in the programming community. In 1983 Biermann, Ballard and Sigmon introduced NLC [12, 13], a natural language notation, which was interpreted directly to an output. In 1984 Knuth proposed *Literate Programming* [14], which combined TEX and Pascal to produce a vocabulary that had the primary goal of documenting for humans what the programmer desires. In 2000 Price, Rilofff, Zachary, and Harvey introduced Natural Java [15]. Natural Java provides a notation that allows the programmer to define a procedure in English, which is converted to Java. This tool allows the programmer to create program structures and edit a selective part of the program with this Natural Language interface. The core program is stored in a Java Abstract Syntax Tree (AST).

Researchers have also considered the application of natural language techniques for creating software artifacts such as requirements documents and source code. There are also efforts to use natural language techniques to analyze artifacts created in conventional programming languages. Michael Ernst suggested using these techniques to analyze all kinds of artifacts [16]. He suggests using this approach with "...error messages, variable names, procedure documentation, and user questions." Similarly, there have been efforts to define the user interface by extracting information from the natural language requirements documents [17]. This approach performs a static analysis on the multiple artifacts to find bugs and generate code. Essentially this approach uses natural language tools and techniques to identify (and possibly satisfy) requirements for the program by analysis of the information that the developer has created to date. In 2001 Overmyer, et al. demonstrated the use of linguistics analysis to convert requirements documents to models of the subject requirements [18]. This approach represents another step along the path to natural language programming.

In another approach [19], Landhaeusser and Hug attempt to use full English to derive program logic. English tends to be verbose, and a programming language based on the entire English language results in significant content being required. Our approach utilizes a Controlled version of English, which results in a simplified syntax. This simplified syntax allows the program to be created with a concise source document.

One of the biggest challenges in mimicking natural language communications with a computer system is the things humans leave unsaid. Much of human interactions are dependent upon shared experience and idioms, which allow humans to provide incomplete information and enable the listener to fill in the rest. Without these implied nuances, human communications would be much more verbose. The challenge for using a controlled natural language for defining a computer program is that we must replicate, at least in part, these techniques which humans use to share information.

Controlled natural languages (CNL) have been applied to many fields of endeavor both within computer science and elsewhere [20]. Many CNL implementations seek to be general to allow their use across multiple areas of study. One well-known CNL of this type is the Attempto Project [21]. The project describes its language as *"...a rich subset of standard English designed to serve as a knowledge representation language."* This tool has been applied to a *Multilingual Semantic Wiki* [22], a Reasoning Engine [23], and a knowledge representation language as used by the Web Ontology Language (OWL) [24]. In these applications, Attempto has provided consistency to the tools by involving a specific subset of English.

Exman et al. [25] offers an interesting tool to translate programming language back into natural language. This tool is intended to allow programmers to understand a previously created program even without a working knowledge of the programming language in question. The application of natural language and machine learning continues to grow in their application to program development [26]. Many of these tools show promise for additional development and application.

2.3 Requirements Capture

The process of requirements capture has been the subject of improvement efforts within the software development community. These efforts have ranged from the rigorous, structured approaches embodied in formal methods and UML to the minimalist approach of *User Stories* utilized by *eXtreme Programming* [27]. Requirements capture is an area where Controlled Natural Language approaches have previously been used [28]. For some years, the agile development community has sought to develop better ways to capture user requirements. *Test-Driven Development (TDD)* [29] was initially associated with agile development in Kent Beck's book on *eXtreme Programming* [30] and then expanded upon in his book on the subject [31]. This methodology seeks to direct the programming effort towards requirements as embodied in a series of tests. These tests are generated by the development team from user requirements. However, they are not in a form that most users could recognize. More recently, parts of the agile community have embraced *Behavior-driven Development (BDD)* as a starting point. *Behavior-driven Development* [32, 33] seeks to describe the user's requirements, which can be converted into tests. These tests are then used as those envisioned in *Test-driven Development*. These requirements are described in a natural language form that can be translated into tests. As such, BDD acts as a front-end for TDD. Cucumber [34, 35] and jBehave [36] are two popular tools that allow developers to capture their requirements in an end-user-friendly format and produce a test suite for TDD application. While these methodologies and associated tools enable the user to describe the requirements in a natural language format, they still require the program to first be created in a traditional programming language.

2.4 Dynamic Programming Languages

In recent years there has been significant growth in the use of dynamic programming languages for mainstream development. While Java and C with their various derivatives continue to be widely used, Python (ranked number one in the TIOBE index), JavaScript (and its derivative, TypeScript), PHP, Ruby, and Perl have moved into the top twenty most popular languages in the TIOBE Index [37] and the Stack-Overflow annual programmer survey [38]. Dynamic programming languages have gained a following because they have helped to improve the productivity of programmers. The combination of dynamic typing and concise syntax results in fewer lines of code being required to achieve the desired result. These advantages have led to claims of productivity gains from 5 to 10 times [39]. With the advent of robust, dynamically typed languages, developers have begun using these tools for applications previously thought to be the domain of traditional statically typed languages. These languages have a syntax that is easier for a programmer to understand, even if written by someone else. In general, the syntax used by these languages is closer to that of a natural language. They still do require conformance to a strict set of rules. However, they have limited the requirements for computer-driven structures like variable declarations, which add to a traditional programming language's verbosity. While these languages' use does not involve automation, they show that other cleaner, simpler syntax languages offer improved programmer productivity opportunities.

2.5 Static Analysis

Static analysis tools come in a range of capabilities. The simplest of these tools are commonly referred to as lint tools [40]. These tools review the program code and identify violations of syntax rules provided for each target programming language. Violations can include punctuation, the misspelling of reserved words, variables that are declared but never used, and other errors that can be identified by reviewing the source code. Static analysis is an area that has seen considerable activity. In addition to stylistic checks, traditionally the approach of linters, these tools have taken more ambitious approaches such as the use of *bug patterns*. Two of the most popular and successful products in the area are FindBug and PMD [41]. They have proved very useful in finding bugs in code that is already written. They help improve the code quality but do not help in the creation of the code.

2.6 Integrated Development Environments

The most used tool for developers is the Integrated Development Environment (IDE). Tools such as Visual Studio, Eclipse, NetBeans, IntelliJ, PyCharm, XCode, and others [42, 43, 44, 45, 46, 47] provide a wide range of features to make the developer more productive. Among these many capabilities is syntax highlighting [48], which involves highlighting various constructs and keywords with colors and formatting to identify their function and usage. These tools can aid the programmer by identifying errors in code when the color coding of the source code does not match their intent. These features also include code completion [49], which automatically completes various words and constructs within the program based on the context and previously entered code. Modern IDEs also provide for the integration of tools such as linters and other static analysis tools. While a modern IDE is a valuable productivity enhancer, it still requires that the programmer code the program in the target programming language's particular syntax.

2.7 Declarative Syntax

Imperative programming [50] is the style utilized by most of the popular programming languages. These languages require the programmer to describe how to construct the various objects that make up a program. To build a user interface, the program would include the tedious steps required to draw each object and then link them to the program logic. This process results in the code being voluminous and difficult to read. It also can obscure the nature of what the programmer is trying to achieve. Listing 2.1 contains the Swift code involved in creating a simple button that invokes a method called *processEachPayThis*. This example includes eleven lines of code. For all but the most knowledgeable, this code is hard to read and obscures the nature of the programmer's goal.

In 2019 Apple introduced SwiftUI [51], which utilizes a declarative syntax for describing the program's user interface. Declarative syntax [52] describes the results the programmer wants to achieve but not how to achieve that result. Listing 2.2 includes the SwiftUI code required to create the same button as captured in Listing 2.1 but does it in seven lines of code, three of which contain only structural

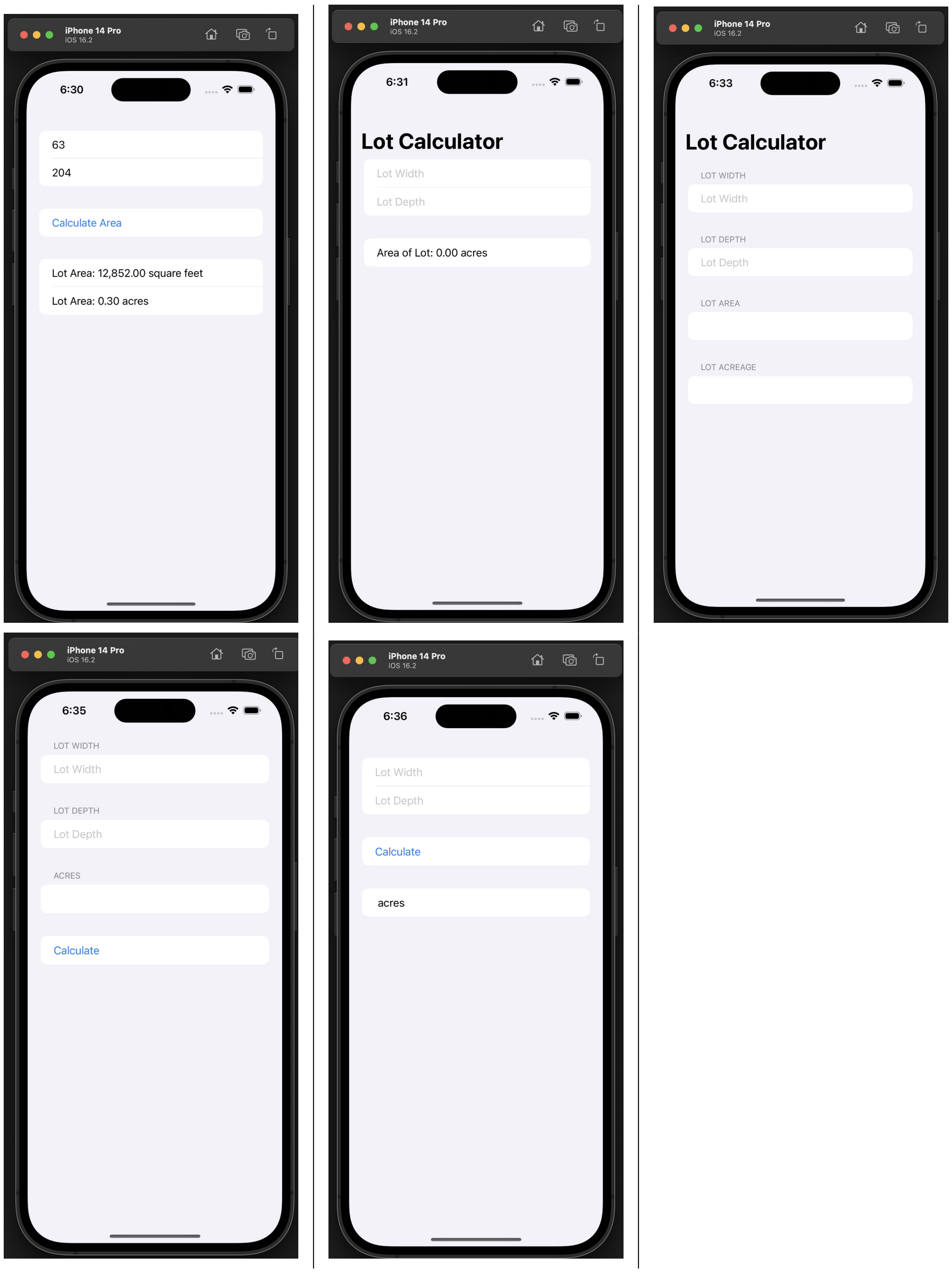
```swift
let button2 = UIButton(type: .system)
button2.setTitle("Calculate", for:.normal)
button2.frame = CGRect(x:self.view.bounds.maxX * 0.0,
                       y:35 * 3,
                       width:self.view.bounds.maxX * 0.5,
                       height:30)
button2.titleLabel?.textAlignment = .left
button2.addTarget(self,
                  action: #selector(processEachPayThis),
                  for: .touchDown)
self.view.addSubview(button2)
```

LISTING 2.1: Swift code for Simple Button.

```swift
HStack {
    Button(action: {
        self.processEachPayThis()
    }) {
        (Text("Calculate"))
    }
}
```

LISTING 2.2: SwiftUI code for Simple Button.

symbols. This code is easier to read and to understand what the programmer is trying to achieve. While this code is considerably simpler than the Swift code, it still is rigid in its syntax and contains numerous special words/commands. It requires the programmer to conform to a strict set of rules. As we describe *CABERNET* in this paper, we will see that it can describe this same button in two lines of code without these strict rules.



iPhone 14 Pro
iOS 16.2
6:30
63
204
Calculate Area
Lot Area: 12,852.00 square feet
Lot Area: 0.30 acres

iPhone 14 Pro
iOS 16.2
6:31
Lot Calculator
Lot Width
Lot Depth
Area of Lot: 0.00 acres

iPhone 14 Pro
iOS 16.2
6:33
Lot Calculator
LOT WIDTH
Lot Width
LOT DEPTH
Lot Depth
LOT AREA
LOT ACREAGE

iPhone 14 Pro
iOS 16.2
6:35
LOT WIDTH
Lot Width
LOT DEPTH
Lot Depth
ACRES
Calculate

iPhone 14 Pro
iOS 16.2
6:36
Lot Width
Lot Depth
Calculate
acres

FIGURE 2.2: Acreage calculators produced by CoPilot.

Chapter 3

Vision

Natural language programming has long been an aspiration for the program development world. Programming languages tend to be complex and idiosyncratic. Scotty, the Chief Engineer from Star Trek, saying "Good Morning computer" to his computer is what people envision as the future of computer interaction. Flying cars are what we have been told the future will include. But are any of these reasonable expectations? By the time we completely describe the function of a program in English, we have a document much longer than your typical computer program. At the same time, there is a significant chance that the English description will be incomplete. Omissions in the English language description of a program are opportunities for errors. If an English language program description is long, complex, and prone to errors what value does it bring? Might we have a better solution if we limit the natural language document to describing only the result and not the details required to get there?

3.1 Describe the result

Much of programming involves describing the process that the computer must go through to achieve the desired goal. This approach allows the programmer broad flexibility in what they can achieve. However, in most cases, the look and feel of the program are dictated by the platform that the programmer is targeting. The types of objects, what they look like and where they are placed on the screen are dictated by a set of platform rules or guidelines. If we incorporate these rules into the development tool or a set of templates used by the development tool; the programmer can avoid having to describe them in the program.

As a result, the program need only describe the unique aspect of each object within the program. These would include unique appearance and behavior characteristics and interactions between the various objects.

3.2 Program housekeeping

In most programs, significant content is dedicated to error checking and other housekeeping activities. These include checking for common errors, such as blank inputs, divide by zero errors and type errors. These processes are well known, and standard methods exist to deal with them. There is little technical challenge in identifying where these processes are needed and how to implement these error-checking and correction methods. Automation of this process can significantly reduce the size of the input program. The added benefit is increasing the readability of the program. The error-checking and correction methods can be verbose and do not seem to relate to the primary processes being implemented. Eliminating this code makes for cleaner application code and code which is more directly related to the function being implemented.

3.3 Flexible vocabulary

The challenge for people learning a programming language is the rigidity of the terminology. Programming languages have a fixed vocabulary that a programmer must learn. Once this vocabulary is learned the programmer must conform to its rules and avoid conflicts between the vocabulary and the object names used in their programs. On the other hand, English and other natural languages are quite flexible in this regard. If our programming approach allows for more flexibility, it will be easier for a developer to move between programming languages. It also allows people with different native languages to adapt to the tool more easily.

Additionally, this flexibility allows the programmer to use terminology which is appropriate to the domain for which the application is intended without having to be concerned about conflicts between the domain language and the programming language.

3.4 Goals

Our goal for *CABERNET* is to incorporate these three visions in our programming tool. We seek to limit the content of the *CABERNET* program to only describing the desired result, provide the required housekeeping tasks in the tool rather than in our program source code and accommodate a flexible vocabulary for our programming tool. This combination will provide a productive development environment. Additionally, we made *CABERNET* adaptable to the programmer. The intent is that the input terminology is easily adjusted and updated. While the approach has been specifically applied to iOS programming in the prototype and examples, the same approach can be applied to other targets.

As summarized in the introduction, we seek to determine if a controlled natural language-based programming tool can provide a highly readable, flexible, extensible, and easy-to-learn development methodology.

Chapter 4

Approach

The core research question which we are addressing is, *"Can a controlled natural language based programming tool provide a highly readable, flexible, extensible, and easy to learn development methodology?"* To that end, we have developed ***CABERNET* (Code generAtion BasEd on contRolled Natural languagE inpuT)**, an approach that allows a programmer to define a computer program using a Controlled Natural Language (CNL). Figure 4.1 lists the key advantages of the *CABERNET* development approach.

- Increased programmer efficiency
- Flexible and straightforward syntax
- English-like (controlled natural language)
- Address needs of all programmers
- Natural language
  - More flexible
  - More forgiving
- Inference fills in gaps

FIGURE 4.1: Key Characteristics of CABERNET

Because the application domain is of limited scope and is specific to the program definition, we can fill in the blanks using inference and implication. This approach provides a result similar to that experienced by typical human communication. We are not seeking to replace the developer in this process. Instead, our work seeks to provide a tool that makes programmers more effective in their efforts while still allowing them to control the process. As an example, we have addressed the challenges of creating a mobile application. This domain has proved challenging for programmers for several reasons including those listed in Figure 4.2.

- Limited screen size
- Multiple possible screen proportions
- Multiple operating systems and widget preferences

FIGURE 4.2: Programming Challenges of Mobile Applications

The use of constraint-based user interface design [53] has helped address these challenges. However, it has added complexity of its own. Constraints must completely define the size and location of features relative to each other and the underlying hardware. At the same time, it must avoid over-constraining the user interface. If a developer is not careful, they may define a set of constraints that work for one device configuration but fail for another. This conflict can nullify the advantages that led us to constraint-based design in the first place. Current techniques include both code-based definition and graphical-based user interface design. Both methods have advantages and disadvantages, but neither has proven to provide an ideal combination of power and productivity. Our approach allows for a flexible description of the application in a natural language notation.

It allows for a minimal description of the user interface, yet it results in a canonical model as output. Since screen size and proportions are handled through templates, the programmer is freed from dealing with those during development. Our approach allows the programmer to define an application for this popular platform with a simple human-friendly approach.

4.1 Basic Principles

The simplicity and directness of the approach are possible because many aspects of the design can be inferred from the context. A programmer developing an application for a mobile device seeks to conform to a set of user interface guidelines. These guidelines become one of the many contextual influences on the application design. As previously noted, one significant advantage enjoyed by humans in their use of natural language is the shared knowledge that allows for portions of the communications to be implied. To overcome this challenge in human-computer communications, we have utilized three techniques.

First, we have used a *broad set of defaults* applied when the developer omits the needed information from their descriptions. Second, we use *inference* to determine the developer's intent from the information provided (both within the user interface description and other artifacts that make up the program). Third, our approach allows *machine learning* to adjust the defaults based on developer choices during the development process. When information is missing or the information provided is ambiguous, we *offer the developer options* from which to choose a solution. Based on these choices and the default solutions that the developer accepts or declines, we build and reinforce our recommended solutions[1]. The characteristics

[1]While the design of *CABERNET* and the *CABERNET* environment support this feature it is not implemented in the prototype at this time. See Section 5.4.1

of the proposed Controlled Natural Language model are listed in Figure 4.3.

4.2 Flexible Nomenclature

One of the challenges of dealing with a natural language is the variety of words or phrases used for a single object or concept. To deal with this, we make use of a thesaurus. We identify a group of words or phrases that can be used interchangeably. Table 4.1 includes some examples of these lists of synonyms.

Widget Type	*Synonyms*
Binary input widget	"option," "switch," "checkbox"
Application	"App," "Application," "Program"
Application Screen	"Window," "Screen," "Scene"
Process Directive	"save," "undo," "calculate," "evaluate"
Switch State	"true," "selected," "on"
Load new screen	"go," "go to," "load," "to"

TABLE 4.1: CABERNET Synonym Examples

These lists are just a small sample of possible synonyms that we should consider. Going one step further, we consider what may be implied by a word or phrase. For example, the last item in the list includes the word "to" and we interpret it to imply "go to." These lists of synonyms are created in several ways. First, they are generated from our knowledge of the domain and the terminology used by programmers. Second, we can expand them using online resources like *thesaurus.com, thesaurus.Babylon-software.com, etc.* Third, we can use search to find terms that are common in the subject area. Finally, we can learn from the developer as they provide feedback when the *CABERNET* processor cannot interpret the term.

4.3 Declarative with a Difference

We have seen the improvement in readability and productivity that is offered by declarative programming approaches like *SwiftUI*. *CABERNET* takes that concept further; it offers declarative with a difference. *CABERNET* combines a declarative style with a natural language-based syntax. It then utilizes inference to discern the programmer's intent. We couple that with a robust set of defaults and templates to convert the program into a native executable.

Figure 4.4 depicts the process of converting a *CABERNET* source into an executable program. The process starts by tokenizing the *CABERNET* source based on the structure of the Markdown outline. The tokenized version is then inspected for terms that can be matched with synonyms in the thesaurus. Where there are tokens that seem to be missing, they are added by inference. The resulting tokens or groups of tokens are identified as actions, symbols, formatting, etc. based on their context. The accuracy of that identification is then tested based on other objects in the program. Where appropriate, outline levels are then simplified using Natural Language tools. The *CABERNET* processor then generates code for the target platform by applying the appropriate templates. Finally, the program is compiled or interpreted by the target platform development tool.

- Input language is forgiving
  - Outline-based structure
  - Flexible
    - Allow use of synonyms, acronyms, and standard abbreviations
    - Allow flexibility in ordering and location of descriptions
  - Terse
    - Minimum input required
    - In most cases, the input is keyword-based and does not require English sentences
    - Each bullet has limited context
  - Utilize popular Markdown [54] lightweight markup language
- Model processing
  - Tool processes natural language model
  - Outputs canonical model
  - Offers alternative interpretations
  - Identifies ambiguous elements
  - Highlights unrecognized and unused elements
- Canonical model
  - Unambiguous
  - Consistent with the natural language model and with itself
  - Can target alternate platforms (iOS, Android, Etc.)
- Tools
  - Predefined rules
  - Learn additional rules from experience
  - Learn from documentation of target framework

FIGURE 4.3: Characteristics of CABERNET CNL

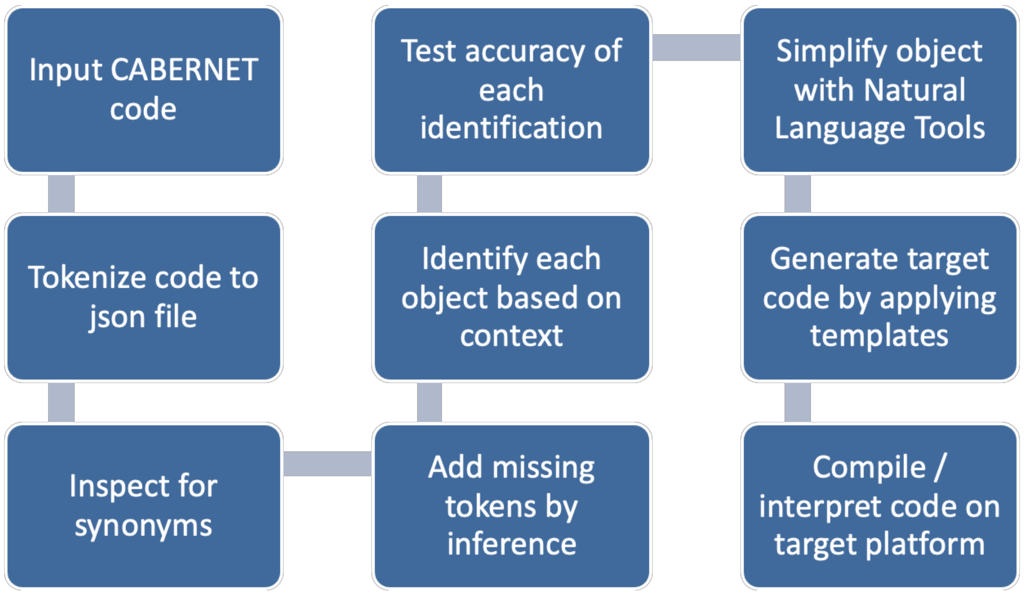

Input CABERNET code
Tokenize code to json file
Inspect for synonyms
Add missing tokens by inference
Identify each object based on context
Test accuracy of each identification
Simplify object with Natural Language Tools
Generate target code by applying templates
Compile / interpret code on target platform

FIGURE 4.4: Processing CABERNET program.

Chapter 5

Notation

5.1 Markdown

The notation for the Controlled Natural Language tool is based on the *Markdown* [55, 56] lightweight markup language. Markdown was created in 2004 by John Gruber [57]. The original intent of Markdown was to provide a tool that allows writers to compose web content in plain text with minimal formatting information. In many ways, the intent of Markdown is like that of *CABERNET* but addresses HTML creation rather than mobile applications. Markdown is intended to focus on the content rather than the formatting information [58]. The same case can be made for *CABERNET*. This makes the use of Markdown as the underlying basis of *CABERNET* predictive.

An additional benefit of Markdown as the underlying format of *CABERNET* is that the source code can be processed using the Markdown tool. The result is an attractively formatted file that displays the program structure without the Markdown tags and formatting characters. By applying a customized CSS style sheet the Markdown processed *CABERNET* can be formatted to improve the readability of the source code.

5.2 Outline Structure

A *CABERNET* program is structured as an outline, including only the information necessary to distinguish itself from the default. Some high-level outline properties that define the *CABERNET* syntax are identified in Figure 5.1.

- #, ##, ###, etc. = Object hierarchy
  - # App = Application
  - ## Scene / Screen
- "*" = Properties and / or actions
  - Object with no properties is a label
  - Properties which begin with a verb = Button
  - "blank", "phone number", etc. = input field
  - "option" = checkbox or switch
- Quoted Text = Literal

FIGURE 5.1: CABERNET Outline Properties

The outline structure captures the hierarchical structure of the program. Each succeeding indentation of the outline represents another embedded structure in the resulting program. The CNL code of an example application is found in Appendix A. Line 1 of this code identifies the basic application. Lines 2 and 31 are one level indented from the application and start two different screens. The lines such as 4, 5, and 7 that begin with '###' are one additional level indented and define the objects on the subject screen. Lines such as 36 that begin with '####' are embedded within the proceeding object or, as in this case, are placed side-by-side on the screen. The decision to embed or to place side-by-side is made based on the context. It would not make sense to embed a button within another button, so the objects are placed side-by-side.

Outline entries that start with a '*' describe the content of the various objects. Entries such as 10, 13, and 27, which contain adjectives, are descriptions of the object's format. Entries such as 35 and 37 that start with a verb describe actions to be taken when clicking the object. Entries like that beginning on line 45 define a calculation that is used to populate the field. Lines like 19 and 25, which do not fall into other categories provide a default entry for the field.

5.3 Thesaurus

The use of a CNL means that multiple names can describe an object in the user interface. For example, in Appendix A, line 2, we refer to one screen of the application as a "Scene." On line 31, we call the second screen as a "Screen." Additionally, these objects can be called different things based on the target platform involved. As a result, the subject tool must create alignment between what the CNL code calls an object and what the target platform expects. To allow *CABERNET* to accommodate this varied nomenclature, we have implemented the concept of a thesaurus. The thesaurus captures a range of words that can be treated as synonyms. Examples of the thesaurus word lists are shown in Figure 4.1. The contents of these thesaurus entries come from the multiple sources shown in Figure 5.2.

The approach utilizes the process shown in Figure 5.3. First, the CNL description is converted from the markdown form to a JSON representation simultaneously, tokenizing key objects in the description. Since the vocabulary utilized in the CNL description can involve multiple names for the same object, we next identify potential synonyms for each object. Then we utilize defaults and templates to fill in the missing parts of the description. In reviewing the program in Appendix A, we can see that, in most cases, the content of the bullets for each object

1. Manually entered values based on our domain experience.
2. Entries from a web-based thesaurus and searches of online documentation.
3. Additional items that *CABERNET* learns from programmers.

FIGURE 5.2: Sources of Synonyms

is very concise. These contain verbs like “calculate” or “cancel” (or “to” which is interpreted to imply “go-to”) or adjectives like “blank,” “italic,” “selected,” “red” or “green” and nouns like “background” or “option.” These verbs, nouns, and adjectives combined with targets like the names of screen objects or other screens make up most of the outline-based program. In cases where a complex function must be executed (like the mathematical calculation in line 45), it is written in a more English-like format. Because the content of each bullet is very concise the opportunity for confusion is limited.

It should be noted that, as in this example, the function can be captured in a flexible combination of English words and mathematical symbols. These statements are processed using natural language processing techniques to derive an executable function. Our tool searches the subject statement to identify the names of objects in the program, such as “Bill amount,” “Tip Percentage,” and “Split.” An entry may contain mathematical symbols and include items that cannot be identified as objects from the program. In this case, it is assumed that the item is not a function but rather text to be displayed. Examples of this include lines 19, 21, and 25.

5.4 Natural Language Processing

As noted, we have limited the description of the application content to the '*' outline levels. Each of these outline items can contain brief entries that describe the content or the material's format. These outline items are also where the controlled natural language entries exist for describing the program function and content. Each item is very limited in scope and context and is therefore relatively easy to interpret. For example, lines 45 through 47 in Appendix A describe the calculation of the value displayed in the object.

> (Multiply Bill amount by Tip Percentage / 100 plus Bill amount) divided by Split

Calculated items like this are identified by mathematical operators' precedents such as *multiply*, *divided by*, *plus*, numbers, and mathematical symbols.

> (***Multiply*** Bill amount by Tip Percentage ***/*** ***100*** ***plus*** Bill amount) ***divided by*** Split

Once an item is identified as potentially being a mathematical calculation, it is further evaluated to see if all the information needed is present to evaluate the item. First, the items are parsed to identify the names of objects in the code that contain the inputs to the calculation. In this example, these include *Bill amount*, *Tip Percentage*, and *Split*.

> (Multiply ***Bill amount*** by ***Tip Percentage*** / 100 plus ***Bill amount***) divided by ***Split***

The remaining text is then examined for adverbs such as *quickly*, *precisely*, and *carefully* and articles such as *the*, *a*, and *an*, which do not add to our understanding of the calculation being performed.

At this point, we should have all the information we need to evaluate the calculation. The biggest challenge to evaluating the remaining text is to understand how to group the calculation. Mathematical expressions are usually evaluated from left to right adjusted by precedence rules and grouping defined by parenthesis. Our tool uses all of these, but it must also consider grouping defined by the natural language of the statement. In its simplest form, this could include *"a times b," "a * b,"* or *"multiply a times b."* All three of these statements are equivalent and do require any special consideration of the grouping of the items. A more complicated example could involve *"(a + b + c) / d," "divide a plus b plus c by d," "divide the sum of a and b and c by d,"* or *"(a plus b + c) divided by d"*. This last example will have a different result than *"a plus b plus c divided by d"* which would be the same as *"a + b + (c / d)"*. By considering the grouping provided by English statements of the forms *"Divide...expression...by...expression"*, *"Multiply...expression...times...expression"* or *"Sum of...expressions,"* we can properly evaluate the calculations described in the natural language of these expressions. In this example, we need to determine to which values the "multiply" at the beginning of the line applies.

(***Multiply*** Bill amount ***by*** Tip Percentage / 100 plus Bill amount) divided by Split

Using the analysis approach we have described, it is clear that *Multiply* goes with *by* so that we multiply *Bill amount* by *Tip Percentage* In evaluating these natural language expressions, *CABERNET* converts these statements to traditional mathematical expressions with the proper grouping of calculations. As a result, *CABERNET* can evaluate traditional mathematical expressions directly if that is what is provided in the program input. In this case, the result is the following expression.

("Bill amount" * "Tip Percentage" / 100 + "Bill amount") / "Split"

If the programmer had entered *Bill* rather than *Bill amount* or *Tip* rather than *Tip Percentage*, we would have failed to complete the transformation. However, this is an example of where we would have prompted the programmer for guidance.

These would be an example of where the transformation was close, and we would have suggested to the programmer a possible match. In some cases, an item will include mathematical symbols or appear to describe a calculation, but *CABERNET* cannot convert it to a mathematical expression. Lines 19, 21, and 25 are examples of this. These lines contain mathematical symbols, but the other text does not contain object names, so we cannot translate them into formulas.

"xxxxx-xxxx" or *"(xxx) xxx-xxxx"* or `‘‘mm/dd/yyyy’’`

Based on the evidence's extent, *CABERNET* will evaluate the likelihood that the programmer's intent is a calculation. If *CABERNET* believes that the item is intended to contain a calculation, it will prompt the programmer for clarification.

A mathematical calculation is but one type of item described in a *CABERNET* outline item. Using the same approach *CABERNET* can evaluate a wide range of program constructs. The steps in the process are as shown in Figure 5.4

This approach can be used for a wide range of programming constructs. By combining items such as database queries, logic statements, mathematical expressions, graphic generation, and file manipulation, we can generate a working program.

5.4.1 Ambiguous Content

Given the issues with natural language processing, there can be more than one interpretation of the CNL form. When *CABERNET* is not able to process a bullet it is interpreted as a text field and added to the app as a String. There will be cases where *CABERNET* detects some evidence for a possible interpretation of a bullet. One example of this is a bullet where the content seems to be a mathematical expression but the processor is not able to identify some of the variables. In these cases, the developer can be presented with options that the tool identifies as the most likely interpretation and possible alternate options. This could include a percentage evaluation of how much of the bullet was successfully interpreted. The results of this selection process are also an opportunity for the system to learn from these decisions. This capability is not implemented in the current prototype but the path to its implementation is understood.

5.4.2 Example Application

Figure 5.5 represents the output of our example application. In Figure 5.5(a), we have the entry screen for an address book application. The App and Scene bullets are for organization and are used to separate the application by screens. “Settings” and “Done” are actions and become buttons. The descriptions of the actions taken for each of the tappable objects are listed as sub-bullets. Next comes multiple blank fields for the contact’s name, company, phone number, email address, and birthday. Finally, there are two option fields represented by switch objects. Depending upon the platform targeted, these could alternately be checkboxes. One of these options is selected by default. Likewise, they could be called switches in the CNL instead of being called options. These alternate names for this object are but one example of how an object can be called multiple things in the CNL or could have multiple objects implemented based on the given CNL. As described above, these choices are made or prioritized based on the developer or target platform preferences.

Figure 5.5(b) is the second screen of the application and includes a tip calculator. This screen contains three blanks filled with the bill amount, the number of people splitting the bill, and the tip percentage. Finally, there is a calculated field representing how much each person pays. As previously described, the text defines this final field in lines 45 through 47 in Appendix A. This calculation is triggered by tapping the “Calculate” button described in lines 34 and 35.

5.5 Advantages and Limitations

Much of the approach's power comes from the flexibility of nomenclature. This flexibility comes from the use of *application-specific dictionaries and thesaurus*, which allow for alternate terms to describe objects and properties within the application. Much of this information is generated based on general domain knowledge. The approach also allows for expanding and customizing this information by *applying search techniques to the target development platform's documentation* / APIs. Using search techniques to index this platform documentation, we can expand and improve the dictionaries and thesaurus used to interpret the CNL input.

Among other advantages, our approach is well suited to *integrate with agile processes*. The CNL source code is *self-documenting* since it is written in a human-readable/understandable form. This human-readable format makes it *easy to understand and refactor* as needed. The result is a dual-purpose artifact (documentation and source code). The implementation is in the form of a domain-specific programming language. Our CNL is not intended to be a general-purpose language like *Attempto English*. As a result, the proposed syntax is concise and lends itself to the proposed application of inference and machine learning. While the example provided in this research involves mobile development, the approach is well-suited for a broad range of programming applications.

While there are many domains where CABERNET is applicable, there are somewhere its dependence upon inference and domain knowledge could be a disadvantage. Within this methodology, we must understand the domain terminology and various synonyms that the programmer may use. For domains not previously addressed, this information may be difficult to come by. We believe that using search techniques to develop a thesaurus for a new domain will help address this limitation.

5.6 Capability of Prototype

The current implementation of *CABERNET* can create multi-screen iPhone and iPad applications. The programs coded in *CABERNET* can be converted to either *Swift* or *SwiftUI* as the intermediate target language. The application can contain labels, text input fields, toggles, and buttons. It can perform mathematical calculations based on inputs provided to the user input fields. The screens are generated based on standard templates for screen layouts. Customization of the generated screens, including colors, fonts, and text styles, is supported. The applications can also include screen navigation, including moving among the screens defined in the application.

The *CABERNET* tool identifies instructions that it is not able to evaluate as simply text entries. It presents those to the programmer but does not provide suggestions for correction (see Section 5.4.1). Since the tool currently only has a limited choice of outputs, these errors are treated as either correct or not.

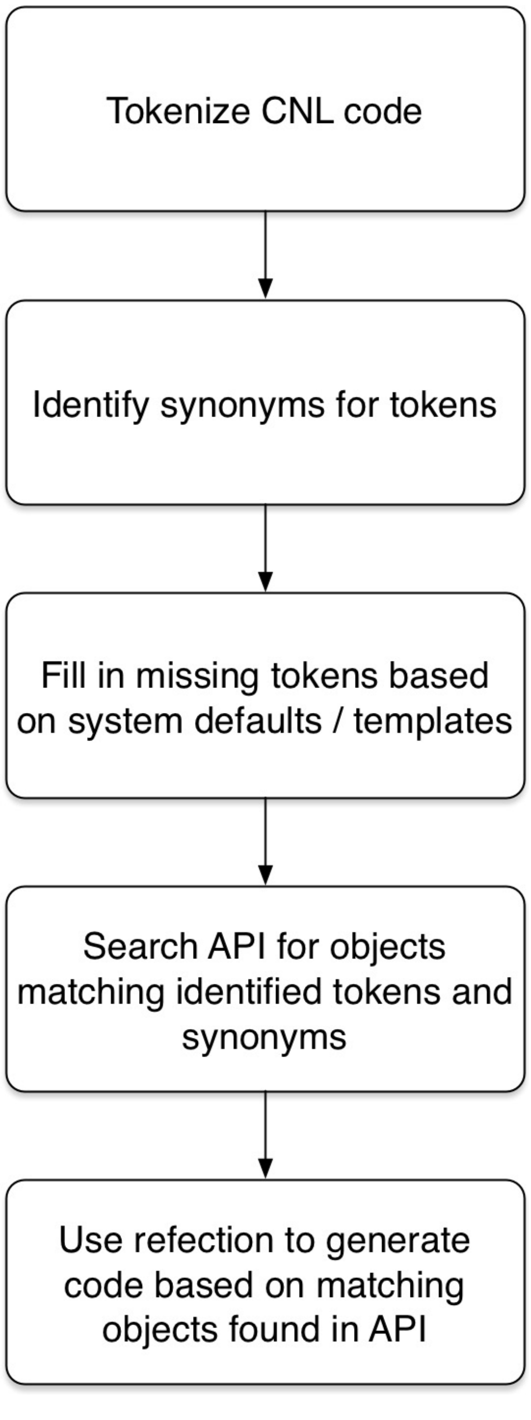

Tokenize CNL code
Identify synonyms for tokens
Fill in missing tokens based on system defaults / templates
Search API for objects matching identified tokens and synonyms
Use refection to generate code based on matching objects found in API

FIGURE 5.3: Interpretation of CNL input.

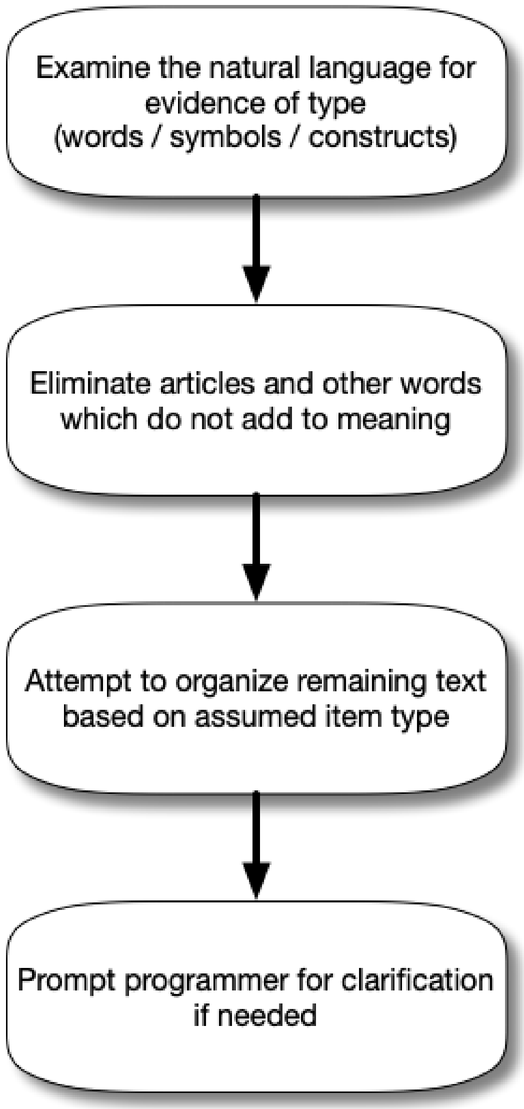
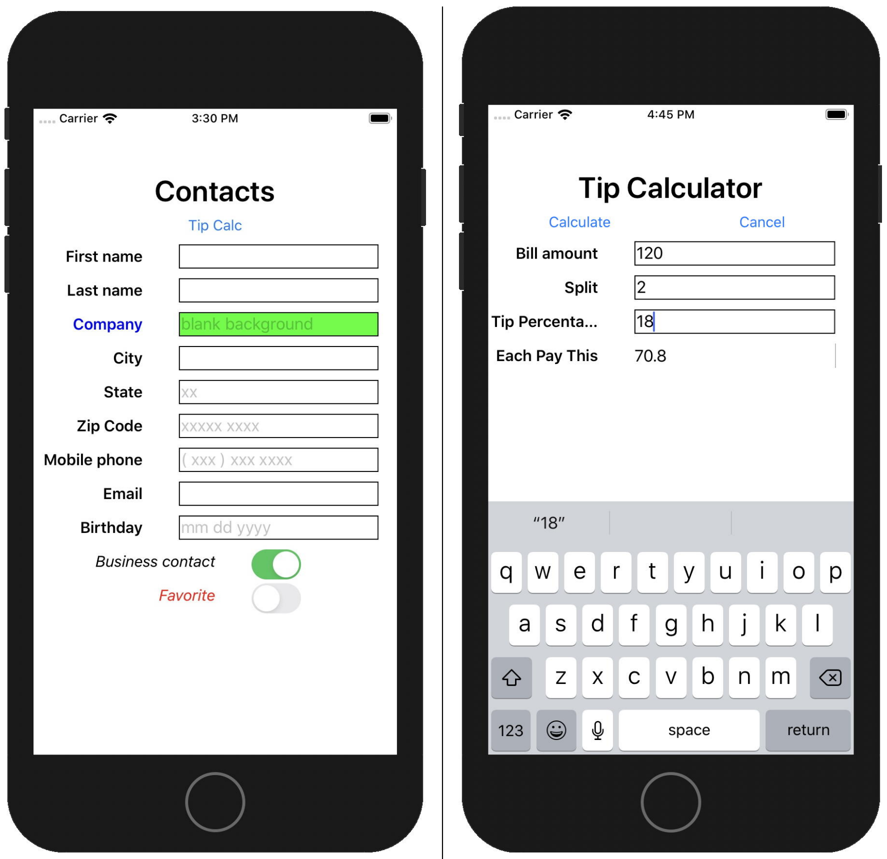

Examine the natural language for evidence of type (words / symbols / constructs)
Eliminate articles and other words which do not add to meaning
Attempt to organize remaining text based on assumed item type
Prompt programmer for clarification if needed

FIGURE 5.4: Construct Processing.


Carrier 3:30 PM

Contacts

Tip Calc

First name	
Last name	
Company	blank background
City	
State	xx
Zip Code	xxxxx xxxx
Mobile phone	(xxx) xxx xxxx
Email	
Birthday	mm dd yyyy

Business contact

Favorite

(a) Addressbook

Carrier 4:45 PM

Tip Calculator

Calculate Cancel

Bill amount	120
Split	2
Tip Percenta...	18
Each Pay This	70.8

(b) Tip Calculator

FIGURE 5.5: App example screens.

Chapter 6

Evaluation

6.1 Research Method

To evaluate the efficacy of *CABERNET* we evaluated it in three ways. First, we compared sample programs written in *CABERNET* with functionally identical programs written in Swift and SwiftUI. This comparison evaluated the size of the programs. Second, we presented these programs to a group of potential users and solicited their feedback on their ability to understand the samples. Third, we asked the users for their feedback on the various programs. This feedback addressed the user's ability to understand the program and their thoughts on how the tools might be used.

6.2 Code Size

The key process metrics that we seek to address with *CABERNET* include *code development speed, clarity, and size*. In Appendix B we have included examples of *CABERNET*, Swift and SwiftUI applications. As can be seen from the examples *CABERNET* programs are very concise. Because they rely heavily on inference, the alignment between how the programmer and the computer understand the program is strong. We can compare the *CABERNET* code for the tip calculator with the alternate ways of implementing this program screen.

Appendices B.1 through B.4 involves a tip calculator application. The 14 lines of code to implement this one screen in *CABERNET* have been included in Appendix B.2. Appendices B.3 and B.4 include the code to implement the same screen in *Swift* and *SwiftUI* respectively. The *Swift* implementation is made up of 122 lines of code. If we eliminate the eight blank lines and ten lines which include only a bracket, both of which are included for readability, the *Swift* implementation still includes 104 lines of code. The *SwiftUI* implementation is made up of 76 lines of code. If we eliminate the two blank lines and seventeen lines that include only a bracket the *SwiftUI* implementation still includes 57 lines of code. Table 6.1 shows a comparison of the code required by each of the languages to create this one screen. As we can see, *Swift* requires over seven times as many lines of code as does *CABERNET* to implement this screen. While the *SwiftUI* implementation is shorter than the *Swift* implementation, it still requires more than four times as many lines of code as does *CABERNET*.

Example		*CABERNET*	*Swift*	*SwiftUI*
Tip Calculator	Lines of Code	14	104	57
	Compared with CABERNET	1X	7.4X	4.1X
Real Estate App	Lines of Code	29	211	96
	Comparison with CABERNET	1X	7.3X	3.3X
Real Estate App	Lines of Code	30		106
Revised	Comparison with CABERNET	1X		3.5X

TABLE 6.1: Comparison of code size.

Much of this *Swift* and *SwiftUI* code implements things that *CABERNET* handles as default values and constructs. That is a large part of what this approach involves. One clear example of this is the actual calculation of the tip value. In the *CABERNET* version, the calculation is defined in line 15. In addition, line 5 describes the action to be taken when we tap the subject button. To perform the same calculation in *Swift*, we need to include lines 4 through 7 to declare the variables involved, lines 23 through 31 to create the button, and lines 107 through 122 to perform the actual calculation. That is a total of twenty-eight lines of code. If we eliminate the eight lines containing only a bracket, we still have twenty lines of code compared with two lines in the *CABERNET* implementation. For the *SwiftUI* implementation, we have lines 4 through 7 for declaring the variables, lines 14 through 20 to define the button, and lines 60 through 71 for the method to perform the calculation. This is a total of twenty-three lines of code. If we again eliminate the lines that contain only brackets, we get sixteen lines of code for this calculation. The *Swift* and *SwiftUI* code must check for common errors like dividing by zero and blank entry fields in addition to the steps required to describe the screen features. *CABERNET* performs these functions by default, thus eliminating the need to check for these things. If there were a reason to allow a program to divide by zero or perform a calculation using an empty field, then *CABERNET* would expect the programmer to say so and describe how it should be handled. In the absence of such descriptions, *CABERNET* assumes that these are errors and handles them appropriately.

The result of these aspects of *CABERNET* is that the source document includes only the basic description of the program content. The implementation, error handling and other processes normally included in a program's source file are all added by the templates and processing done by the *CABERNET* tool. The result is that the *CABERNET* file is brief and easy to understand.

A second example application is included in Appendices B.5 through B.11. This example is a Real Estate application that involves two screens. This application is implemented with a single *CABERNET* file which is 29 lines long (Appendix B.7). This same application takes two files and 211 lines of code in Swift (Appendices B.8 and B.9) and 96 lines of code in SwiftUI (Appendices B.10 and B.11).

The third example involves modifying the Real Estate application to highlight one of the input fields based on the content of that field. The implementation of this example in *CABERNET* and SwiftUI is included in Appendices B.12 and B.13 (note the SwiftUI solution also needs the Acreage Calculator file from Appendix B.11). Like the previous two examples, *CABERNET* required significantly less code to make this addition than SwiftUI. In this case, *CABERNET* required only 30 lines of code and SwiftUI required 106 (including the files in Appendices B.13 and B.11).

Across these examples, Swift required about 7.3 times as many lines of code as *CABERNET*. In the same examples, SwiftUI required between 3.3 and 4.1 times as many lines of code as *CABERNET*. This is a significant difference that results in more opportunities for typos and errors to be introduced. At this point, we should note that *CABERNET* is more forgiving with the input provided. As previously noted, *CABERNET* allows a significant range of word selection in its programs. On the other hand, Swift and SwiftUI require strict adherence to the program structure. The combination of longer programs and strict rules make Swift and SwiftUI more vulnerable to errors.

6.3 Easy to Understand

Lines of code are but one means of measuring the effort required to create a program. Additional measurements involve how difficult it is to craft the code, how readable the code is, and how well the program processing the code deals with alternative inputs. These all contribute to how easy it is for a programmer to learn the language. The measurement of these aspects of the language is more subjective than the simple counting of lines of code. Nevertheless, they are all important to understanding how successful *CABERNET* is / can be in improving programmer productivity.

To understand the relative ease of understanding a program written in *CABERNET* vs. the same program written in Swift or SwiftUI we surveyed 47 people. These survey participants were solicited from undergraduate and graduate students at DePaul University and Virginia Tech computer science programs. When asked about their level of programming ability 31 participants self-identified as a "Student", 7 as a "Developer" and 1 as a "Novice". When asked about their years of programming experience 33 reported 3 or more years of experience and 6 reported 2 or fewer.

The survey participants were provided sample programs implemented in *CABERNET*, Swift and SwiftUI. These samples are included in Appendix B. The survey questions and results can be found in Appendix C. Of the 35 questions included in the survey, there are 8 which ask the participants to evaluate how *Easy to Understand* the various samples were. The results of these questions are included in Figure 6.1. The figure graphs the percentage of responses at a given rating on a scale of zero to ten with ten being the easiest to understand. 80% of the ratings on *CABERNET* were a 7 or better. On the same basis, the Swift and SwiftUI examples were 46% and 52% were rated 7 or better respectively. The actual number of responses for each language is included in Table 6.2.

The mean score for *CABERNET* on these questions was 7.75. The mean score for Swift and SwiftUI were 5.53 and 6.26 respectively. Figure 6.2 shows the normal distribution of the responses for three languages. However, these normal distribution charts can be a bit misleading. The chart in Figure 6.3 shows the actual responses for *CABERNET* and SwiftUI. From this, you can see that while the SwiftUI results form a bell curve around its mean the *CABERNET* results have a more single-sided distribution. 48% of the responses for *CABERNET* are a rating of either a 9 or 10.

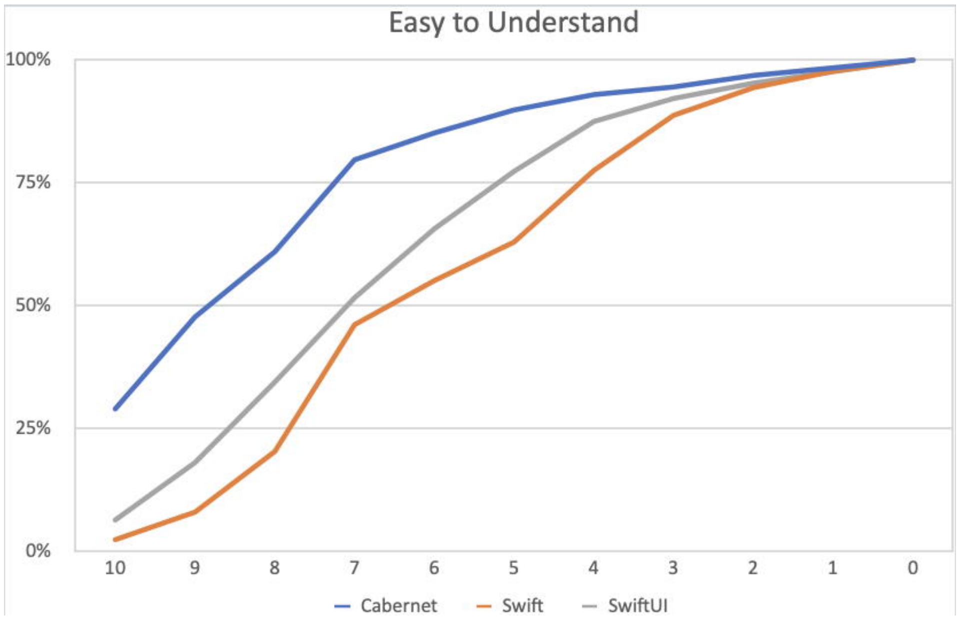


FIGURE 6.1: Ease of Understanding.

Language	10	9	8	7	6	5	4	3	2	1	0
CABERNET	37	24	17	24	7	6	4	2	3	2	2
Swift	2	5	11	23	8	7	13	10	5	3	2
SwiftUI	8	15	21	22	18	15	13	6	4	3	3

TABLE 6.2: Ease of Understanding.

If we consider the groups that self-identify as Students and Developers independently, we get comparable results. The developers gave the *CABERNET* examples a mean score of 8.29 on the *Easy-to-Understand* questions. The students gave *CABERNET* a mean score of 7.62 on these same questions. On the other hand, the developers gave Swift and SwiftUI mean scores of 4.64 and 5.81 respectively. The students gave Swift and SwiftUI mean scores of 5.69 and 6.35 respectively. All these values can be seen in Table 6.3.

Language	Overall	Developers	Students
CABERNET	7.75	8.29	7.62
Swift	5.53	4.64	5.69
SwiftUI	6.26	5.81	6.35

TABLE 6.3: Mean Ease of Understanding Scores.

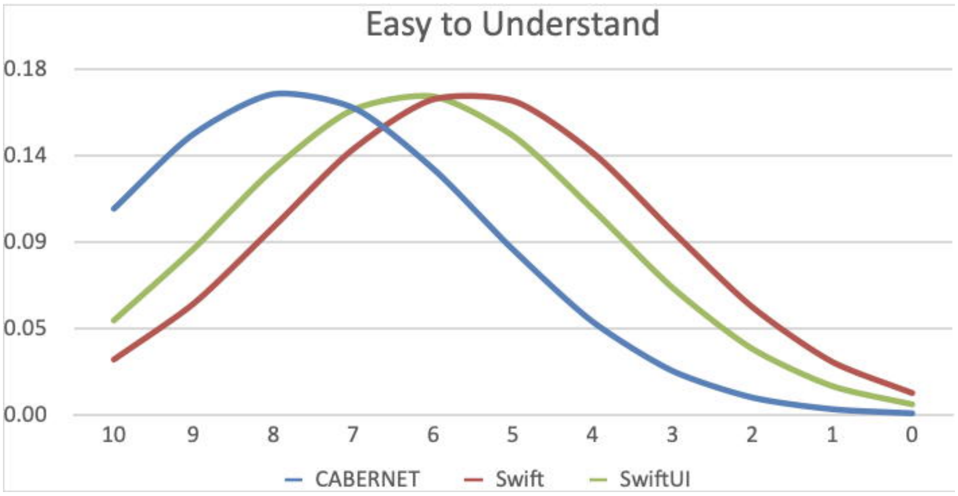


FIGURE 6.2: Normal Distribution of Responses.

6.4 Confidence of Understanding

There were also 8 questions on the survey which ask participants their confidence they understood the code examples and/or do they feel they could code similar structures. A clear understanding of the code is required if the programmers are to modify the examples or create similar programs on their own. Figure 6.4 shows the

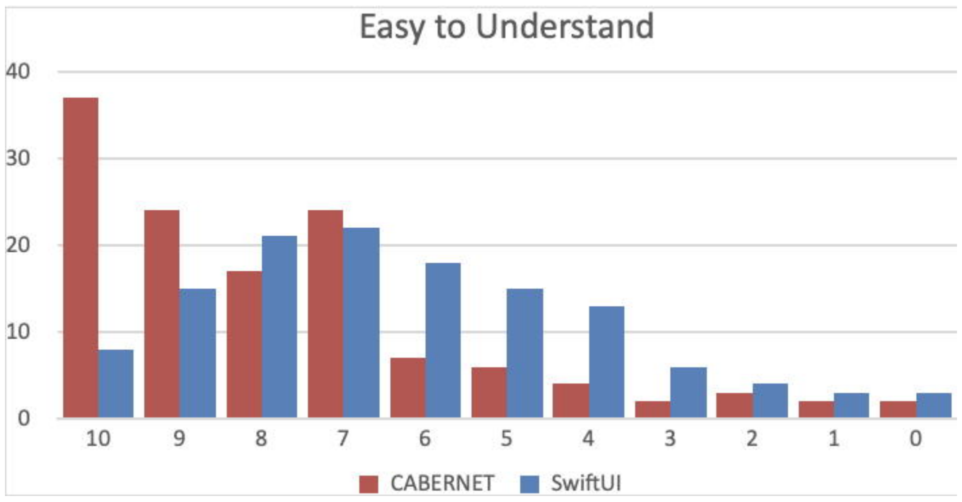


FIGURE 6.3: CABERNET Responses vs SwiftUI.

cumulative number of responses rating the respondent's confidence that they understood the various examples. The respondents gave a mean score on their confidence in understanding the various examples of 6.75, 5.80 and 6.30 for *CABERNET*, Swift and SwiftUI respectively.

From these survey results, we find that the respondents found *CABERNET* clearly easier to understand. Additionally, they are confident they understand how the examples work and would be able to code similar structures.

6.5 Respondent Feedback

In addition to the quantitative responses based on examples, application survey respondents were offered the opportunity to provide general comments about the various programming options. There were two questions included in the survey to this end. These specific questions were as follows:

> Do you have any general comments, observations, or suggestions on the Cabernet tool?

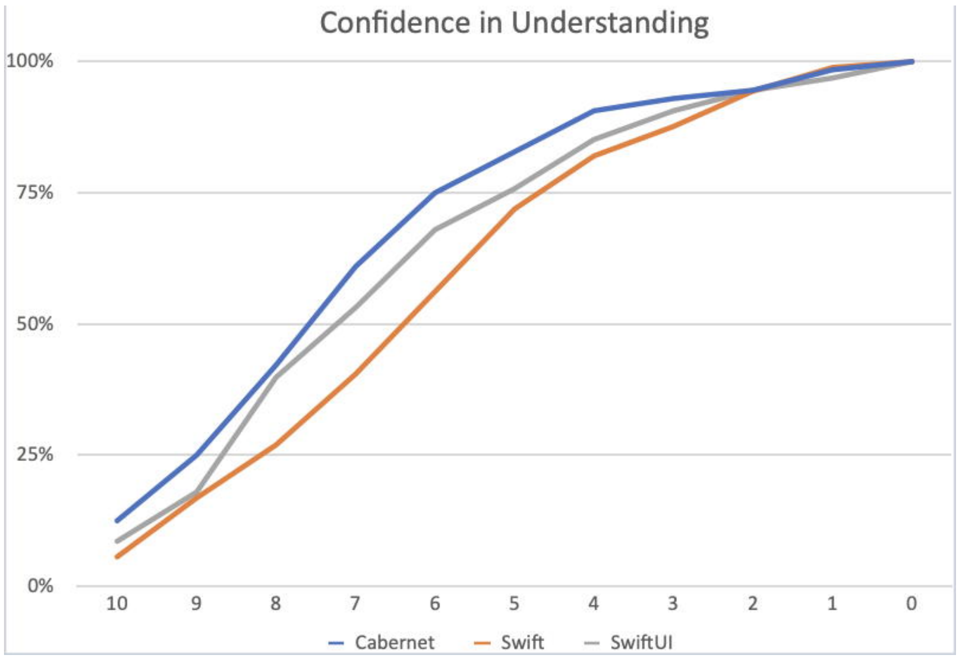


FIGURE 6.4: Confidence of Understanding.

> How would you compare the three development tools used in these examples?

The responses provided to these two questions are included in Appendix D. In total, there were 52 comments submitted. Five of these responses had to do with the mechanics of the survey itself rather than the tools. This leaves 47 comments about the tools. Of the remaining comments, 21 were positive comments about *CABERNET*. 17 of the comments expressed concern about the granularity of control provided by *CABERNET*. A couple of representative examples of these types of comments are as follows.

> "It is much more readable in terms of figuring out what it is doing and judging what the result will look like. However, it seems harder if I

> wanted to make something specific, because I wouldn't know where to start with getting the right syntax."
>
> "I think this would be a good tool for quick form or mockup creation, but there are many things I wonder about it. Like for these examples - can I change size of entry boxes? Can I move fields on the screen? How would a more complicated function look? I am intrigued but scared since so much of the "brains" dictating things is hidden."
>
> "Cabernet is good for a quick solution. The other two are good if you want more specific options and to understand the development tools."

These respondents were concerned that they would not be able to achieve precise control over the end application. In a couple of cases, they equated this with the approach being more suitable for end-user programming. While it is possible to write the requirements for precision control of the resulting application the respondent seemed to want more surety that they know how *CABERNET* will interpret their input.

6.6 Quantitative Results

In Section 6.2 we described a detailed comparison of the code size for *CABERNET*, Swift and SwiftUI for some example applications. This comparison demonstrated a significant difference in the size of the program required for these examples (see Table 6.1). As can be seen, the Swift and SwiftUI implementations require between over 3 and 7 times as much code respectively. As previously discussed, this not only affects the direct productivity of the developer but also affects the potential for errors.

This size difference is a direct result of the basic architecture of the *CABERNET* approach. First, most of the screen layout in a *CABERNET* application is provided by the *CABERNET* tool. This greatly simplifies the source code. Second, most of the error checking for the subject application is generated by *CABERNET* during the process of converting from the source code to the native program code.

This results in a 70 to 85% reduction in source code while automatically including the appropriate error-checking code. This virtually eliminates the possibility of divide by zero, missing input and other common programming errors.

6.7 Qualitative Results

One of the major goals of this research is to create a tool that is easy to understand. As we have seen from the developers and students surveyed *CABERNET* scored very well in this respect (see Section 6.3). On average on a scale of 1 to 10 *CABERNET* received a score of 7.75 for easy to understand. This is very favorable when compared with an average score of 5.53 and 6.26 for Swift and SwiftUI, respectively. This is even more apparent when we see that 80% of the easy-to-understand scores for *CABERNET* were 7 or higher. This compares to 46% and 52% of the scores for Swift and SwiftUI respectively were 7 or higher. These results were consistent for both the group identifying as students and those identifying as developers.

Being easy to understand can be interpreted as being limited to reading someone else's code. It is also important that a developer be able to use the tool. To do this requires that they have confidence they understand the code and can reproduce applications. As we saw from the survey results, (see Section 6.4) the survey respondents had that confidence. They gave *CABERNET* an average score of 6.75 compared with 5.80 and 6.30 for Swift and SwiftUI respectively. So, they found the code significantly easier to understand and they were confident of their understanding.

Chapter 7

Discussion

7.1 State of Research

The effort to date includes the development of the *CABERNET* prototype as described above. Additionally, we have completed the comparison of that prototype against Swift and SwiftUI development tools. This comparison is described in this document. The feedback from survey participants has been generally favorable (see Section 6.5).

One area of concern expressed by the survey participants involved the precision and granularity of control provided to the developer. 36% of the comments expressed concern about being able to control the program result with precision. This makes sense since *CABERNET* allows a programmer to make use of the flexible nomenclature to generate the program. Traditional programming languages provide a limited vocabulary with a single result for each input. Since *CABERNET* allows for multiple alternate inputs, this one-to-one relationship no longer exists. Additionally, since *CABERNET* depends upon templates to generate the end program, there is the question of how the programmer can deal with variations from those templates. This has been a common theme since the beginning of computer programming. Early programming attempts involved direct machine language input. As programming languages evolved, the developer had less direct control over the details of the application. The tension between the productivity offered by continually higher-level programming tools and the precise control offered by lower-level tools has defined the progress of program development methodologies.

Some of the survey comments suggested that *CABERNET* might be best suited for application prototyping or end-user programming. These are understandable suggestions. Both of these domains can benefit from rapid application development. They also can benefit from the simplicity that *CABERNET* provides without being limited by the limited precision of control provided.

7.2 Preliminary Work

I have published several papers that document some of the progress of this effort. All of this research has focused on improving developer efficacy. Early research involved the use of type inference to detect anomalies in dynamic language programs. This approach showed promise and was effective in identifying errors in dynamic language-based programs. This paper, titled *Anomaly detection in dynamic programming languages through heuristics based type inference*, was presented at the 2017 Computing Conference [59].

Subsequent work has taken the use of inference and applied it to a controlled natural language-based programming approach as described in this proposal. I have published two papers on this work. The first of these papers, titled *Code Generation Based on Inference and Controlled Natural Language Input*, I presented at the 6th International Conference on Software Engineering (SOFE 2020) [60]. Subsequently, a paper covering additional progress on this research, titled *Programmer Productivity Enhancement Through Controlled Natural Language Input*, was published in the International Journal of Software Engineering and Applications [61].

My most recent paper titled *Code Generation Based on Controlled Natural Language Input* has been accepted for presentation and subsequent publication at the *8th International Conference on Software Engineering (SOEN 2023)*. This paper covers the current state of this work. This conference will take place in Toronto, Canada on July 22 through 23, 2023.

7.3 Future Direction

The described approach represents just one implementation of a family of languages that can address a wide range of programming environments. Because we are approaching the specific needs of program development, we can avoid the complexity of general-purpose natural language implementations. Search and reflection allow for multiple and new targets without having to generate all the background equivalences. The approach lends itself to growth and evolution to address new and more difficult challenges.

While the existing research is focused on mobile device development, this approach has a much broader potential application. It is easy to see how this same approach can be applied to any number of GUI-based development targets. Beyond that, this overall approach can be utilized for any number of development applications. The combination of outline-based input with each outline bullet handling a single issue of interest can be applied to any number of applications. Combining that with a simplified natural language syntax and allowing for programmer feedback makes the potential number of applications very broad.

The approach results in a lower cognitive barrier for new programmers. At the same time, it allows for significant productivity improvements for experienced developers. The *simplicity and natural form* of the tool make the development process much more approachable. By utilizing templates, synonyms, a search of API, and reflection methodologies, this approach allows for CNL-based code to generate applications for other platforms.

As we pursued this research, we envisioned utilizing search techniques to harvest the required information from new domains and application targets. This is a potential goal for the future. Our ability to accomplish this is dependent upon the quality and organization of the documentation for the target domain. As the quality of tools for understanding natural language, and input improves this becomes a more realistic goal. In reviewing the documentation of Swift and SwiftUI we have found that it is more suitable for human consumption than for programmatically extracting the required information. As a result, we have elected to not implement this as part of the current effort.

Chapter 8

Related Work

8.1 Programmer Productivity

The underlying goal of our research is to improve the productivity of program developers. Of course, the first challenge is to define what we mean by productivity. We view productivity as the quantity of defect-free functionality a developer can produce per unit of time or effort. How do we evaluate that productivity? It is a common belief that productivity varies between program developers by as much as 10:1. While program developer skills affect productivity, it is but one factor in determining the process's productivity. Lutz Prechelt [62] highlights the wide variety of things that affect the program development process's overall productivity. The choice of programming language [63] is a significant element in determining the productivity of the overall process. These evaluations involve comparing traditional languages like C and Java vs. scripting languages like Python and Perl. This work found that the scripting languages resulted in shorter programs and shorter development efforts. At the same time, the run-time performance did not suffer as a result of using scripting languages. These comparisons involve programs that do not involve GUI development. On the other hand, our effort does involve GUI, but we believe the conclusions are the same.

In his research, William Nichols [64] showed that the relationship between programmers and productivity was weak. He found a high degree of variability in programmer productivity across a range of tasks. In comparing developers' performance on a range of tasks, only half of the variation could be attributed to the differences between programmers. If programmer skill is not the overwhelming predictor of development, then we must consider other things when attempting to improve the productivity of the development process.

8.2 Next Paradigm Programming Languages

Yannis Smaragdakis [65] considered how next-generation programming languages will change to support significant productivity improvements. This research is based on the author's experiences developing using DataLog (a declarative language based on ProLog). His conclusions are heavily influenced by the belief that future languages will depend upon the compiler (or interpreter) to perform the heavy lifting behind the scenes. The programmer will specify their desired result in the programming language, and the tool (compiler or interpreter) will determine the methods required to achieve those goals. This conclusion aligns with our approach for *CABERNET*.

Pane, Ratanamahatan, and Myers [66] studied how non-programmers describe problems and how programming languages expect them to be described. This study documents an understanding of things that future programming languages can utilize to try and reduce that gap. It provides background information from which we can draw in the development of *CABERNET*.

Jia-Jun Li et al. [67] offers a generalized approach to end-user-development, which combines natural language programming with *Programming by Demonstration*. This approach allows end-users to demonstrate behavior by recording interactions with example applications. The *PUMICE* tool presented in this research is powerful in the conversion of the demonstration into a generalized user application. However, this tool is intended for the automation of end-user tasks, not application development. This approach requires the user to provide an application to use as the demonstration platform. This limits the scope of development to interactions with existing or similar applications. *CABERNET*, on the other hand, allows for the development of new concepts and allows the developer to provide clarification where required because of feedback from the development tool.

8.3 Natural Programming Languages

The use of natural language input as a programming language has long been envisioned. To date, none of these efforts have been accepted by mainstream programming applications. Good and Howland [68] explored the use of natural languages for teaching programming or computational thinking. This research involves a study of the role-playing game toolkit for Neverwinter Nights 2. The program as shipped allows the creation of scenarios using NWScript, an Electron tool-set-based programming tool. The researchers studied users' programming with NWScript and then using natural-language-based input. They evaluated the ability of non-programmers to script events using NWScript and natural language. They found that none of the users were able to script their events with the NWScript tool successfully. When using unconstrained natural language, they found significant confusion about how to formulate input. After several iterations of more constrained input methods, their final solution involved a hybrid graphical-textual-based programming tool. Their conclusion was the unconstrained natural-language-based programming was not successful for these non-programmers. They concluded that it was necessary to utilize the graphical tool's enhanced structure and provide meaningful feedback to the programmer throughout the coding process. Our approach also recognizes that unconstrained natural language can be confusing for users. However, rather than taking the hybrid approach proposed by Good and Howland, we chose to implement a more focused application of natural language, which allows us to make inferences by the context of the individual natural language phrases.

Gao [69] presents a survey of Controlled Natural Languages (CNL) used for machine-oriented applications. This work includes consideration of Attempto Controlled English (ACE), Processable English (PENG,) and Computer-processable English (CPL). This paper demonstrates the capability of these languages to describe a system of logic. From this research, it is clear that these CNLs make their inputs very constrained and impose a rigid set of rules. These limitations are necessary to enable direct translation to machine-processable logic. The result is a much less natural syntax that imposes rules not that dissimilar to traditional programming languages. At the same time, the resulting languages that are suited to expressions of logic are less appropriate for describing the actions involved in a computer program. Our application, on the other hand, is looking to describe computer applications.

Nadkarni et al. [70] present a technique for converting English-like algorithms into C code. Like our proposal, this approach allows for synonyms and can learn from personalized training with individual programmers. However, their algorithms' syntax is more like pseudo-code than English (see example in Listing 8.1). Like NaturalJava [15] and NLP (Natural Language Processing) for NLP (Natural

Language Programming) [71], the input for this approach is still very programming-language-like and of limited scope.

```
define integer rem
input an integer n
rem = n mod 2
if rem equal to 0 then
end if
else
display 'Given number is odd'
end else
```

LISTING 8.1: Example of "Semi-Natural Language Algorithm to Programming Language Interpreter".

Wang, Ginn, et al. [72] have applied the concept of a language that learns from the programmers to Natural Language input. This approach is conceptually similar to our vision for *CABERNET*. The programmer can specify the notation they want to use to describe various actions that the program needs to take in their approach. In this way, the program compiler/interpreter continually learns from the programmer to the point where most of the programs in their research were based on this user-defined notation. Their research involves game-based programming where users provide instructions for building objects. This work demonstrates how natural language programming can be effective when it grows based on user input. The work to date is limited in scope but does help show direction for the future. Our application is much broader in scope as we are using *CABERNET* to create full programs. In *CABERNET*, we start with a natural language interpreter and allow that interpreter to grow and improve based on programmer input, much like the *Dependency-based Action Language (DAL)* in Wang, Ginn et al.'s research.

8.4 Code Snippets

One of the most common processes used by developers is searching online development resources to identify approaches to solving specific problems. If productivity is the number of program functions that a developer produces per unit of time, then improving that developer's access to example code and solutions to specific programming problems is a means of improving their productivity. StackOverflow is a site frequented by many programmers seeking to find answers to their programming questions. Yan et al. [73] created a tool called *CosBench*, which combines natural language input with a dataset of previous projects and other resources to provide programmers with answers to their programming questions. CosBench takes natural language input and searches for code snippets that are relevant to the search criteria. They compared their results with six other tools attempting to do the same thing. In a survey article, Allamanis et al. [74] identified a depth of work undertaking this same approach. This provides an interesting tool to find solutions to programming problems but is not a programming methodology in itself.

Chapter 9

Conclusions

9.1 Conclusion

9.1.1 Contribution to Computer Science

Natural-language-based programming has long been envisioned as a desirable approach to program development. As discussed, this is an approach that has been previously tried but has not been effectively executed. This research envisioned an implementation of a natural-language-based technique that could be utilized as a general-purpose programming tool.

This work has shown that there is potential for a CNL-based programming tool. *CABERNET* has demonstrated a programming approach that is easy for both developers and students to understand. The tool is significantly more concise than the present common programming techniques for mobile device development. It also incorporates common error-checking techniques without burdening the developer with their implementation.

9.1.2 Results

The described approach involves the use of a Controlled Natural Language to create programs. The code is *human readable and self-documenting* and lends itself to modern agile programming methodologies. Using defaults and templates, the required code is *succinct and allows the system to fill in many of the details*. The result is a very terse source code with detail only included where needed to provide information for the program execution. Our approach significantly simplifies the required code *reducing the code size by a factor of up to 7 to 1.* Not only is there significantly less code, but the content of the program is significantly easier to read and understand. Developers and students that have reviewed examples of these applications have expressed concerns about the granularity of control the programming tool provides. Based on the review and understanding of this feedback, we believe this technique may be most suitable for application in development prototyping and end-user programming.

While this technique is not a solution for all problems it does provide significant benefits for the right problem space. This is a win/win proposition.

- Shorter programs
- Easier to read
- Flexible syntax
- Collaborative interaction with programmer
- Learns from programmer feedback

In summary, a valuable development tool for programmers of all capability levels.

Appendix A

Code of Example Application

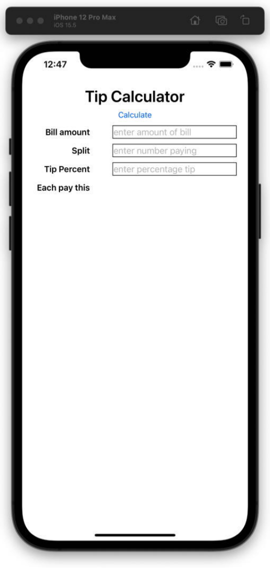
```
# App
## Scene
    * home
### "Contacts"
### "Tip Calc"
    * to calc
### "First name"
    * blank
### "Last name"
    * blank
### "Company"
    * blank
    * background green, blue
### "City"
    * blank
### "State"
    * xx
### "Zip Code"
    * xxxxx-xxxx
### "Mobile phone"
    * (xxx) xxx-xxxx
### "Email"
    * blank
### "Birthday"
```

```
    * mm/dd/yyyy
### "Business contact"
    * option selected
### "Favorite"
    * option
    * italic, red
## Screen
    * calc
### "Tip Calculator"
### "Calculate"
    * calculate Each Pay This
#### "Cancel"
    * cancel entry
### "Bill amount"
    * blank
### "Split"
    * blank
### "Tip Percentage"
    * blank
### "Each Pay This"
    * ( Multiply Bill amount by Tip Percentage / 100 plus Bill amount
        ) divided by Split
```

Appendix B

Examples for Survey

B.1 Example 1, Tip Calculator, Screenshot



B.2 Example 1, Tip Calculator, CABERNET Source Code

```
# App
## Scene
    * tipcalc
### "Tip Calculator"
### "Calculate"
    * calculate Each Pay This
### "Bill amount"
    * enter amount of bill
### "Split"
    * enter number paying
### "Tip Percentage"
    * enter percentage tip
### "Each Pay This"
    * ( Multiply Bill amount by Tip Percentage / 100 plus Bill amount
        ) divided by Split
```

B.3 Example 1, Tip Calculator, Swift Source Code

```swift
import UIKit

class TipcalcView: UIViewController {
    var field3: UITextField!
    var field4: UITextField!
    var field5: UITextField!
    var field6: UILabel!
    override func viewDidLoad() {
        super.viewDidLoad()
        view.isOpaque = true
        view.backgroundColor = .white

        let label1 = UILabel(frame:CGRect(x:0,
                                          y:35 * 1,
                                          width:self.view.bounds.maxX
                                              * 1,
                                          height:100))
        label1.text = "Tip Calculator"
        label1.textAlignment = .center
        label1.font = .boldSystemFont(ofSize:30.0)
        label1.isHighlighted = true
        self.view.addSubview(label1)

        let button2 = UIButton(type: .system)
        button2.setTitle("Calculate", for:.normal)
        button2.frame = CGRect(x:self.view.bounds.maxX * 0.0,
                               y:35 * 3,
                               width:self.view.bounds.maxX * 0.5,
                               height:30)
        button2.titleLabel?.textAlignment = .left
        button2.addTarget(self, action: #selector(processEachPayThis),
            for: .touchDown)
        self.view.addSubview(button2)

        let label3 = UILabel(frame:CGRect(x:self.view.bounds.maxX * 0.0,
                                          y:35 * 4,
                                          width:self.view.bounds.maxX * 0.3,
                                          height:25))
        field3 = UITextField(frame:CGRect(x:self.view.bounds.maxX * 0.4,
                                          y:35 * 4,
                                          width:self.view.bounds.maxX * 0.55,
                                          height:25))
        label3.text = "Bill amount"
        label3.textAlignment = .right
        label3.textColor = .black
        label3.font = .boldSystemFont(ofSize:16.0)
        field3.placeholder = "enter amount of bill"
        field3.backgroundColor = .white
        field3.borderStyle = .line
        self.view.addSubview(field3)
        self.view.addSubview(label3)

        let label4 = UILabel(frame:CGRect(x:self.view.bounds.maxX * 0.0,
                                          y:35 * 5,
                                          width:self.view.bounds.maxX * 0.3,
                                          height:25))
        field4 = UITextField(frame:CGRect(x:self.view.bounds.maxX * 0.4,
                                          y:35 * 5,
                                          width:self.view.bounds.maxX * 0.55,
                                          height:25))
        label4.text = "Split"
        label4.textAlignment = .right
        label4.textColor = .black
        label4.font = .boldSystemFont(ofSize:16.0)
        field4.placeholder = "enter number paying"
        field4.backgroundColor = .white
        field4.borderStyle = .line
        self.view.addSubview(field4)
        self.view.addSubview(label4)

        let label5 = UILabel(frame:CGRect(x:self.view.bounds.maxX *
            0.0,
                                          y:35 * 6,
                                          width:self.view.bounds.maxX
                                              * 0.3,
                                          height:25))
        field5 = UITextField(frame:CGRect(x:self.view.bounds.maxX *
            0.4,
                                          y:35 * 6,
                                          width:self.view.bounds.maxX
                                              * 0.55,
                                          height:25))
        label5.text = "Tip Percent"
        label5.textAlignment = .right
        label5.textColor = .black
        label5.font = .boldSystemFont(ofSize:16.0)
        field5.placeholder = "enter percentage tip"
        field5.backgroundColor = .white
        field5.borderStyle = .line
        self.view.addSubview(field5)
        self.view.addSubview(label5)

        let label6 = UILabel(frame:CGRect(x:self.view.bounds.maxX *
            0.0,
                                          y:35 * 7,
                                          width:self.view.bounds.maxX
                                              * 0.3,
                                          height:25))
        field6 = UILabel(frame:CGRect(x:self.view.bounds.maxX * 0.4,
                                      y:35 * 7,
                                      width:self.view.bounds.maxX *
                                          0.55,
                                      height:25))
        label6.text = "Each pay this"
        label6.textAlignment = .right
        label6.textColor = .black
        label6.font = .boldSystemFont(ofSize:16.0)
        field6.backgroundColor = .white
        self.view.addSubview(field6)
        self.view.addSubview(label6)
    }

    @IBAction func doDismiss(_ sender: Any?) {
        self.presentingViewController?.dismiss(animated: false)
    }
    @objc func processEachPayThis() {
        if let field3_text = field3.text {
            if let field3_float = Float(field3_text){
                if let field5_text = field5.text {
                    if let field5_float = Float(field5_text){
                        if let field4_text = field4.text {
                            if let field4_float = Float(field4_text){
                                field6.text = String((field6_float*
                                    field5_float/100+field3_float)/
                                    field4_float)
                            }
                        }
                    }
                }
```

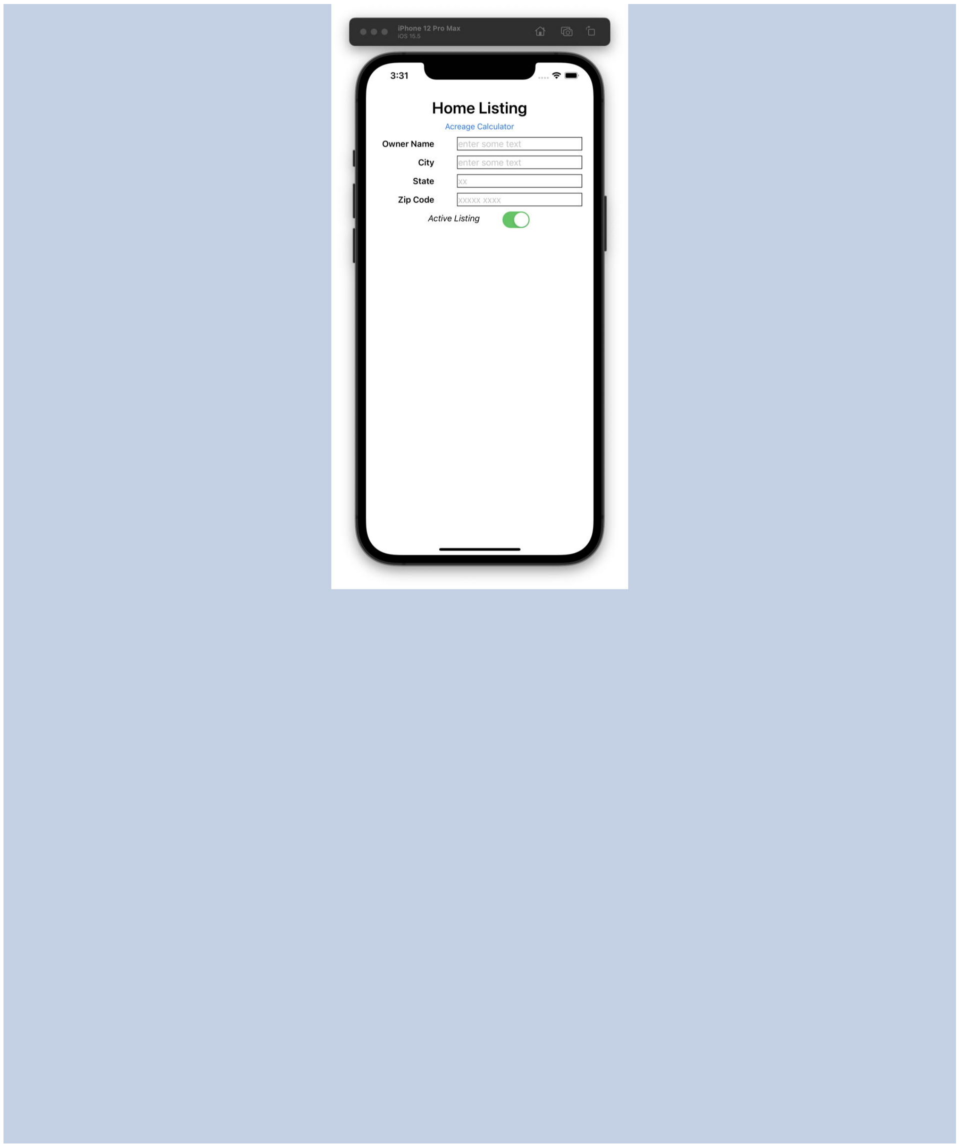
```
                  }
            }
      }
}
```

B.4 Example 1, Tip Calculator, SwiftUI Source Code

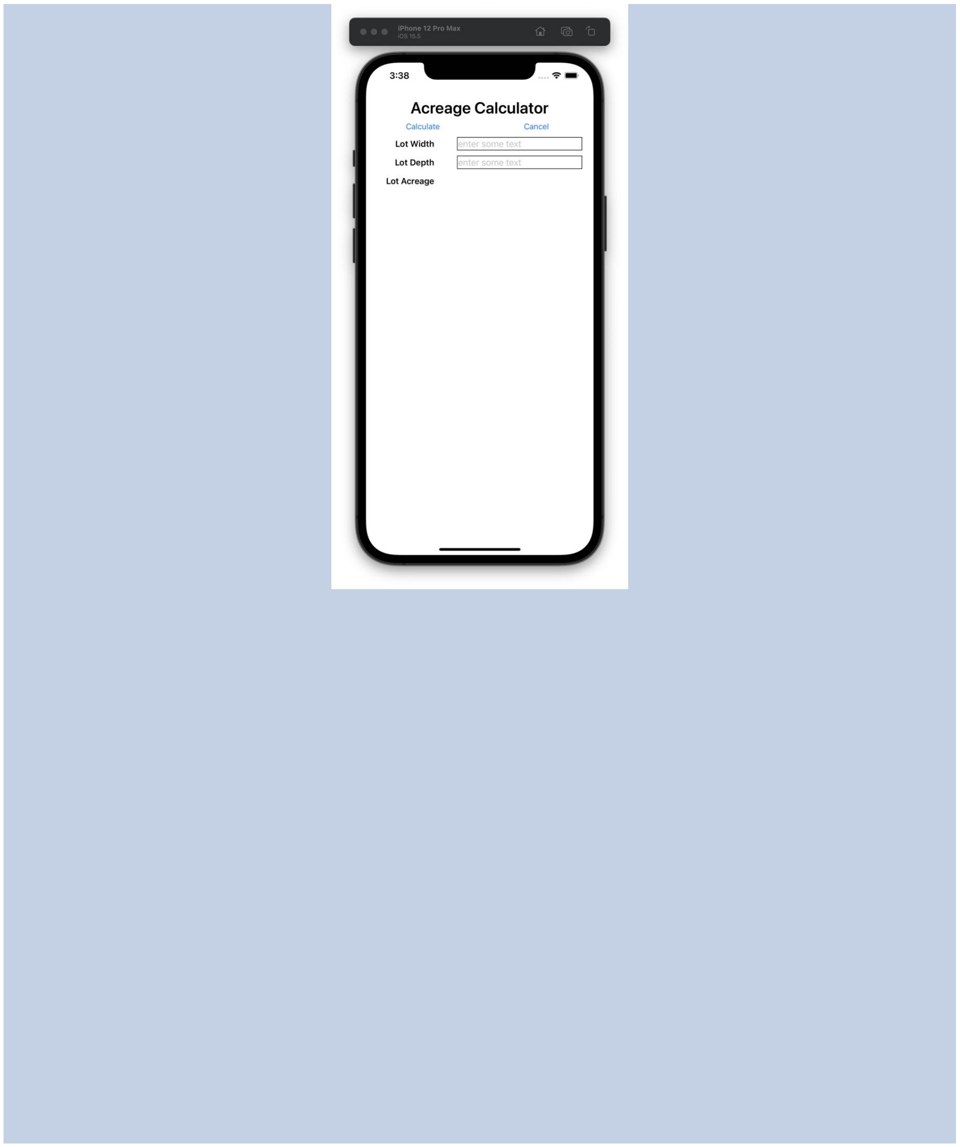
```swift
import SwiftUI

struct TipcalcView: View {
    @State var field3: String = ""
    @State var field4: String = ""
    @State var field5: String = ""
    @State var field6: String = ""
    var body: some View {
        NavigationView {
            VStack {
                VStack {
                    Text("Tip Calculator")
                        .font(.largeTitle)
                    HStack {
                        Button(action: {
                            self.processEachPayThis()
                        }) {
                            (Text("Calculate"))
                        }
                    }
                    HStack {
                        Text("Bill amount")
                        Spacer()
                        TextField("enter amount of bill", text:
                            $field3)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("Split")
                        Spacer()
                        TextField("enter number paying", text: $field4)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("Tip Percent")
                        Spacer()
                        TextField("enter percentage tip", text: $field5)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("Each pay this")
                        Spacer()
                        TextField("", text: $field6)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .disabled(true)
                    }
                }
                Spacer()
            }
            .padding()
        }
    }

    private func processEachPayThis() {
        let int_field3 = Int(field3) ?? 0
        let int_field5 = Int(field5) ?? 0
        let int_field4 = Int(field4) ?? 0
        if int_field4 != 0 {
```

```swift
            field6 = String((int_field3 * int_field5 /  100  +
                int_field3) / int_field4)
        }
        else {
            field6 = ""
        }
    }
}
struct TipcalcView_Previews: PreviewProvider {
    static var previews: some View {
        TipcalcView()
    }
}
```

B.5 Example 2, Real Estate App, Screenshot, page 1 of 2



B.6 Example 2, Real Estate App, Screenshot, page 2 of 2



B.7 Example 2, Real Estate App, CABERNET Source Code

```
# App
## Scene
    * home
### "Home Listing"
### "Acreage Calculator"
    * to acregecalc
### "Owner Name"
    * enter some text
### "City"
    * enter some text
### "State"
    * xx
### "Zip Code"
    * xxxxx-xxxx
### "Active Listing"
    * option selected
## Screen
    * acreagecalc
### "Acreage Calculator"
### "Calculate"
    * calculate Lot Acreage
### "Cancel"
    * cancel entry
### "Lot Width"
    * enter some text
### "Lot Depth"
    * enter some text
### "Lot Acreage"
    * Multiply Lot Width by Lot Depth / 43560
```

B.8 Example 2, Real Estate App, Swift Source Code

```
import UIKit

class ContentView: UIViewController {
    var field3: UITextField!
    var field4: UITextField!
    var field5: UITextField!
    var field6: UITextField!
    var switch7: Bool!
    override func viewDidLoad() {
        super.viewDidLoad()
        view.isOpaque = true
        view.backgroundColor = .white

        let label1 = UILabel(frame:CGRect(x:0,
                                          y:35 * 1,
                                          width:self.view.bounds.maxX
                                             * 1,
                                          height:100))
        label1.text = "Home Listing"
        label1.textAlignment = .center
        label1.font = .boldSystemFont(ofSize:30.0)
        label1.isHighlighted  = true
        self.view.addSubview(label1)

        let button2 = UIButton(type: .system)
        button2.setTitle("Acreage Calculator", for:.normal)
        button2.frame = CGRect(x:self.view.bounds.maxX * 0.0,
                               y:35 * 3,
                               width:self.view.bounds.maxX * 1,
                               height:30)
        button2.titleLabel?.textAlignment = .left
        button2.addTarget(self, action: #selector(gotoAcreagecalcView)
            , for: . touchDown)
        self.view.addSubview(button2)

    let label3 = UILabel(frame:CGRect(x:self.view.bounds.maxX * 0.0,
                                      y:35 * 4,
                                      width:self.view.bounds.maxX *
                                          0.3,
                                      height:25))
    field3 = UITextField(frame:CGRect(x:self.view.bounds.maxX * 0.4,
                                      y:35 * 4,
                                      width:self.view.bounds.maxX *
                                          0.55,
                                      height:25))
    label3.text = "Owner Name"
    label3.textAlignment = .right
    label3.textColor = .black
    label3.font = .boldSystemFont(ofSize:16.0)
    field3.placeholder = "enter some text"
    field3.backgroundColor = .white
    field3.borderStyle = .line
    self.view.addSubview(field3)
    self.view.addSubview(label3)

    let label4 = UILabel(frame:CGRect(x:self.view.bounds.maxX * 0.0,
                                      y:35 * 5,
                                      width:self.view.bounds.maxX *
                                          0.3,
                                      height:25))
    field4 = UITextField(frame:CGRect(x:self.view.bounds.maxX * 0.4,
                                      y:35 * 5,
                                      width:self.view.bounds.maxX *
                                          0.55,
                                      height:25))
    label4.text = "City"
    label4.textAlignment = .right
    label4.textColor = .black
    label4.font = .boldSystemFont(ofSize:16.0)
    field4.placeholder = "enter some text"
    field4.backgroundColor = .white
    filed4.borderStyle = .line
    self.view.addSubview(field4)
    self.view.addSubview(label4)

    let label5 = UILabel(frame:CGRect(x:self.view.bounds.maxX * 0.0,
                                  y:35 * 6,
                                  width:self.view.bounds.maxX *
                                     0.3,
                                  height:25))
    field5 = UITextField(frame:CGRect(x:self.view.bounds.maxX * 0.4,
                                  y:35 * 6,
                                  width:self.view.bounds.maxX *
                                     0.55,
                                  height:25))
    label5.text = "State"
    label5.textAlignment = .right
    label5.textColor = .black
    label5.font = .boldSystemFont(ofSize:16.0)
    field5.placeholder = "xx"
    field5.backgroundColor = .white
    field5.borderStyle = .line
    self.view.addSubview(field5)
    self.view.addSubview(label5)

    let label6 UILabel(frame:CGRect(x:self.view.bounds.maxX * 0.0,
                                y:35 * 7,
                                width:self.view.bounds.maxX * 0.3,
                                height:25))
    field6 = UITextField(frame:CGRect(x:self.view.bounds.maxX * 0.4,
                                  y:35 * 7,
                                      width:self.view.bounds.maxX *
                                        0.55,
                                      height:25))
    label6.text = "Zip Code"
    label6.textAlignment = .right
    label6.textColor = .black
    label6.font = .boldSystemfont(ofSize:16.0)
    field6.placeholder = "xxxxx xxxx"
    field6.backgroundColor = .white
    field6.borderStyle = .line
    self.view.addSubview(field6)
    self.view.addSubview(label6)

    let label7 = UILabel(frame:CGRect(x:self.view.bounds.maxX * 0.0,
                                      y:35 * 8,
                                      width:self.view.bound.maxX *
                                        0.5,
                                      height:25))
    let switch7 = UISwitch(frame:CGRect(x:self.view.bounds.maxX * 0.6,
                                        y:35 * 8,
                                        width:self.view.bounds.maxX *
                                          0.35,
                                        height:25))
    label7.text = "Active Listing"
    label7.textAlignment = .right
    label7.textColor = .black
    label7.font = .italicSystemFont(ofSize:16.0)
    switch7.isOn = true
    self.view.addSubview(switch7)
    self.view.addSubview(label7)

    }

    @IBAction func doDismiss(_ sender: Any?) {
        self.presentingViewController?.dismiss(animated: false)
```

```swift
    }

    @objc func gotoAcreagecalcView() {
        let svc = AcreagecalcView(nibName: nil, bundle: nil)
        svc.modalPresentationStyle = .fullScreen
        self.present(svc, animated: false)
    }

}
```

B.9 Example 2, Real Estate App, Swift Acreage Calculator Source Code

```
import UIKit

class AcreagecalcView: UIViewController {
    var field4: UITextField!
    var field5: UITextField!
    var field6 UILabel!
    override func viewDidLoad() {
        super.viewDidLoad()
        view.isOpaque = true
        view.backgroundColor = .white

        let label1 = UILabel(frame:CGRect(x:0,
                                          y:35 * 1,
                                          width:self.view.bounds.maxX
                                              * 1,
                                          height:100))
        label1.text = "Acreage Calculator"
        label1.textAlignment = .center
        label1.font = .boldSystemFont(ofSize:30.0)
        label1.isHighlighted = true
        self.view.addSubview(label1)

        let button2 = UIButton(type: .system)
        button2.setTitle("Calculate", for:.normal)
        button2.frame = CGRect(x:self.view.bounds.maxX * 0.0,
                               y:35 * 3,
                               width:self.view.bounds.maxX * 0.5,
                               height:30)
        button2.titleLabel?.textAlignment = .left
        button2.addTarget(self, action: #selector(processLotAcreage),
            for: .touchDown)
        self.view.addSubview(button2)

        let button3 = UIButton(type: .system)
        button3.setTitle("Cancel", for:.normal)
        button3.frame = CGRect(x:self.view.bounds.maxX * 0.5,
                               y:35 * 3,
                               width:self.view.bounds.maxX * 0.5,
                               height:30)
        button3.titleLabel?.textAlignment = .left
        button3.addTarget(self, action: #selector(doDismiss(_:)),for:
            .touchDown)
        self.view.addSubview(button3)

        let label4 = UILabel(frame:CGRect(x:self.view.bounds.maxX *
            0.0,
                                          y:35 * 4,
                                          width:self.view.bounds.maxX
                                              * 0.3,
                                          height:25))
        field4 = UITextField(frame:CGRect(x:self.view.bounds.maxX *
            0.0,
                                          y:35 * 4,
                                          width:self.view.bounds.maxX
                                              * 0.55,
                                          height:2))
        label4.text "Lot Width"
        label4.textAlignment = .right
        label4.textColor = .black
        label4.font = .boldSystemFont(ofSize:16.0)
        field4.placeholder = "enter some text"
        field4.backgroundColor = .white
        field4.borderStyle = .line
        self.view.addSubview(field4)
        self.view.addSubview(label4)

        let label5 = UILabel(frame:CGRect(x:self.view.bounds.maxX *
            0.0,
                                          y:35 * 5,
                                          width:self.view.bounds.maxX
                                              * 0.3,
                                          height:25))
        field5 = UITextField(frame:CGRect(x:self.view.bounds.maxX *
            0.4,
                                          y:35 * 5,
                                          width:self.view.bounds.maxX
                                              * 0.55,
                                          height:25))
        label5.text = "Lot Depth"
        label5.textAlignment = .right
        label5.textColor = .black
        label5.font = .boldSystemFont(ofSize:16.0)
        field5.placeholder = "enter some text"
        field5.backgroundColor = .white
        field5.borderStyle = .line
        self.view.addSubview(field5)
        self.view.addSubview(label5)

        let label6 = UILabel(frame:CGRect(x:self.view.bounds.maxX *
            0.0,
                                          y:35 * 6,
                                          width:self.view.bounds.maxX
                                              * 0.3,
                                          height:25))
        field6 = UILabel(frame:CGRect(x:self.view.bounds.maxX * 0.4,
                                      y:35 * 6,
                                      width:self.view.bounds.maxX *
                                          0.55,
                                      height:25))
        label6.text = "Lot Acreage"
        label6.textAlignment = .right
        label6.textColor = .black
        label6.font = .boldSystemFont(ofSize:16.0)
        field6.backgroundColor = .white
        self.view.addSubview(field6)
        self.view.addSubview(label6)

    }

    @IBAction func doDismiss(_ sender: Any?) {
        self.presentingViewController?.dismiss(animated: false)
    }
    @objc func processLotAcreage() {
        if let field4_text = field4.text {
            if let field4_float = Float(field4_text){
                if let field5_text = field5.text {
                    if let field5_float = Float(field5_text){
                        field6.text = String(field4_float*field5_float
                            /43560)
                    }
                }
            }
        }
    }
}
```

B.10 Example 2, Real Estate App, SwiftUI Source Code

```swift
import SwiftUI

struct ContentView: View {
    @State var field3: String = ""
    @State var field4: String = ""
    @State var field5: String = ""
    @State var field6: String = ""
    @State var switch7: Bool = false
    var body: some View {
        NavigationView {
            VStack {
                VStack {
                    Text("Home Listing")
                        .font(.largeTitle)
                    HStack {
                        NavigationLink(destination: AcreagecalcView())
                            {
                            Text("Acreage Calculator")
                        }
                    }
                    HStack {
                        Text("Owner Name")
                        Spacer()
                        TextField("enter some text", text: $field3)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("City")
                        Spacer()
                        TextField("enter some text", text: $field4)
```

```
                            .frame(width: 240)
                            .multilineTextAligment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("State")
                        Spacer()
                        TextField("xx", text: $field5)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("Zip Code")
                        Spacer()
                        TextField("xxxxx xxxx", text:$field6)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
                    }
                    Toggle(isOn: $switch7){
                        Text("Active Listing")
                    }
                }
                Spacer()
            }
            .padding()
        }
    }
}
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

B.11 Example 2, Real Estate App, SwiftUI Acreage Calculator Source Code

```swift
import SwiftUI

struct AcreagecalcView: View {
    @State var field4: String = ""
    @State var field5: String = ""
    @State var field6: String = ""
    var body: some View {
        NavigationView {
            VStack {
                VStack {
                    Text("Acreage Calculator")
                        .font(.largeTitle)
                    HStack {
                        Button(action: {
                            self.processLotAcreage()
                        }) {
                            (Text("Calculate"))
                        }
                    }
                    HStack {
                        Text("Lot Width")
                        Spacer()
                        TextField("enter some text", text: $field4)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("Lot Depth")
                        Spacer()
                        TextField("enter some text", text: $field5)
```

```
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("Lot Acreage")
                        Space()
                        TextField("", text: $field6)
                            .frame(width: 240)
                            multilineTextAlignment(.center)
                            .disabled(true)
                    }
                }
                Spacer()
            }
            .padding()
        }
    }

    private func processLotAcreage() {
        let float_field4 = Float(field4) ?? 0
        let fload_field5 = Float(field5) ?? 0
        field6 = String(float_field4 * float_field5 / 43560)
    }
}
struct AcreagecalcView_Previews: PreviewProvider {
    static var previews: some View {
        AcreagecalcView()
    }
}
```

B.12 Example 3, Real Estate App with Logic, CABERNET Source Code

```
# App
## Scene
    * home
### "Home Listing"
### "Acreage Calculator"
    * to acregecalc
### "Owner Name"
    * enter some text
    * when equals " " background yellow
### "City"
    * enter some text
### "State"
    * xx
### "Zip Code"
    * xxxxx-xxxx
### "Active Listing"
    * option selected
## Screen
    * acreagecalc
### "Acreage Calculator"
### "Calculate"
    * calculate Lot Acreage
### "Cancel"
    * cancel entry
### "Lot Width"
    * enter some text
### "Lot Depth"
    * enter some text
### "Lot Acreage"
    * Multiply Lot Width by Lot Depth / 43560
```

B.13 Example 3, Real Estate App with Logic, SwiftUI Source Code

```swift
import SwiftUI

struct ContentView: View {
    @State var field3: String = ""
    @State var field4: String = ""
    @State var field5: String = ""
    @State var field6: String = ""
    @State var switch7: Bool = false
    var body: some View {
        NavigationView {
            VStack {
                VStack {
                    Text("Home Listing")
                        .font(.largeTitle)
                    HStack {
                        NavigationLink(destination: AcreagecalcView()) {
                            Text("Acreage Calculator")
                        }
                    }
                    if field3 == "" {
                        HStack {
                            Text("Owner Name")
                            Spacer()
                            TextField("enter some text", text: $field3)
                                .frame(width: 240)
                                .multilineTextAlignment(.center)
                                .border(Color.black)
                                .background(Color.yellow)
                        }
                    }
                    else {
                        HStack {
                            Text("Owner Name")
                            Spacer()
                            TextField("enter some text", text: $field3
                                )
                                .frame(width: 240)
                                .multilineTextAlignment(.center)
                                .border(Color.black)
                        }
                    }
                    HStack {
                        Text("City")
                        Spacer()
                        TextField("enter some text", text: $field4)
                            .frame(width: 240)
                            .multilineTextAligment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("State")
                        Spacer()
                        TextField("xx", text: $field5)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
                    }
                    HStack {
                        Text("Zip Code")
                        Spacer()
                        TextField("xxxxx xxxx", text:$field6)
                            .frame(width: 240)
                            .multilineTextAlignment(.center)
                            .border(Color.black)
```

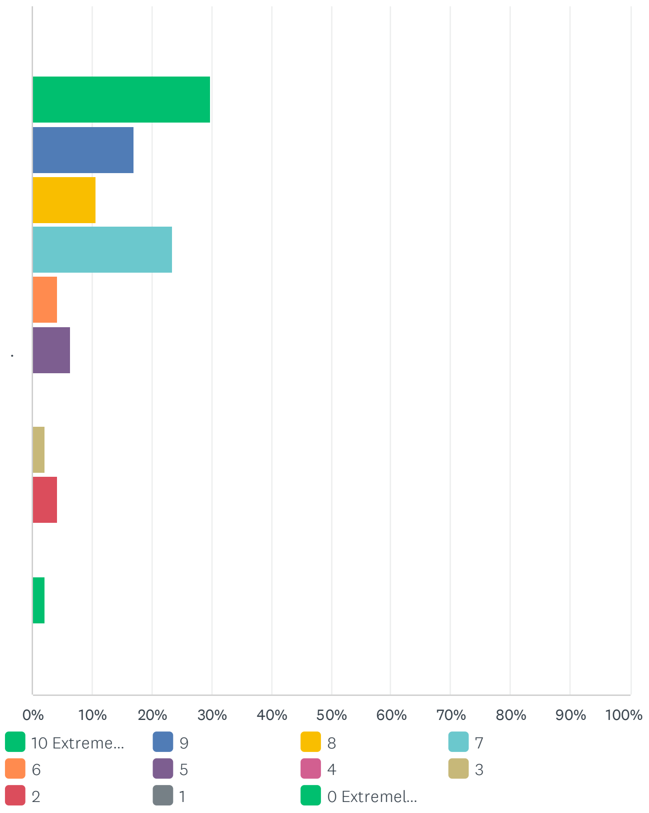
```swift
                    }
                    Toggle(isOn: $switch7){
                        Text("Active Listing")
                    }
                }
                Spacer()
            }
            .padding()
        }
    }
}
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Appendix C

Results of Programmer Survey

Q1 Using any number from 0 to 10, where 0 is extremely difficult and 10 is extremely easy, what number would you use to rate how easy it was for you to understand the Cabernet source code?

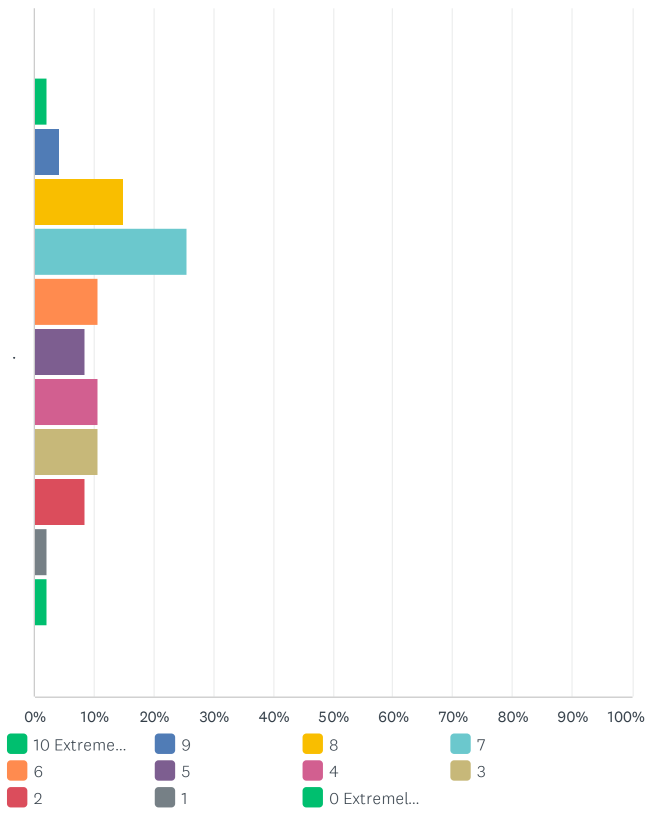
Answered: 47 Skipped: 0



	10 EXTREMELY EASY	9	8	7	6	5	4	3	2	1	0 EXTREMELY DIFFICULT	TOTAl
.	29.79% 14	17.02% 8	10.64% 5	23.40% 11	4.26% 2	6.38% 3	0.00% 0	2.13% 1	4.26% 2	0.00% 0	2.13% 1	4

Q2 Using any number from 0 to 10, where 0 is extremely difficult and 10 is extremely easy, what number would you use to rate how easy it was for you to understand the Swift source code?

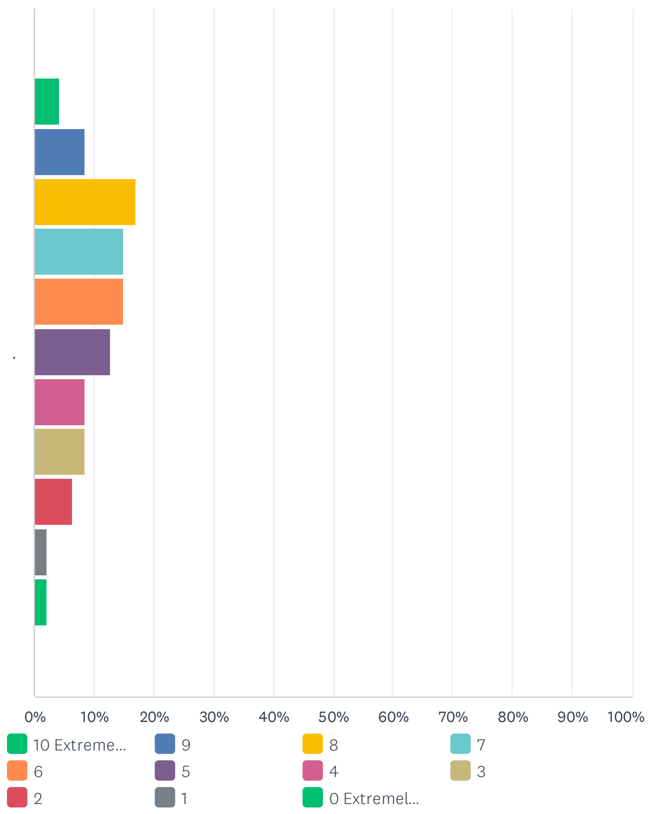
Answered: 47   Skipped: 0



	10 EXTREMELY EASY	9	8	7	6	5	4	3	2	1	0 EXTREMELY DIFFICULT	TOT
.	2.13% 1	4.26% 2	14.89% 7	25.53% 12	10.64% 5	8.51% 4	10.64% 5	10.64% 5	8.51% 4	2.13% 1	2.13% 1	

Q3 Using any number from 0 to 10, where 0 is extremely difficult and 10 is extremely easy, what number would you use to rate how easy it was for you to understand the SwiftUI source code?

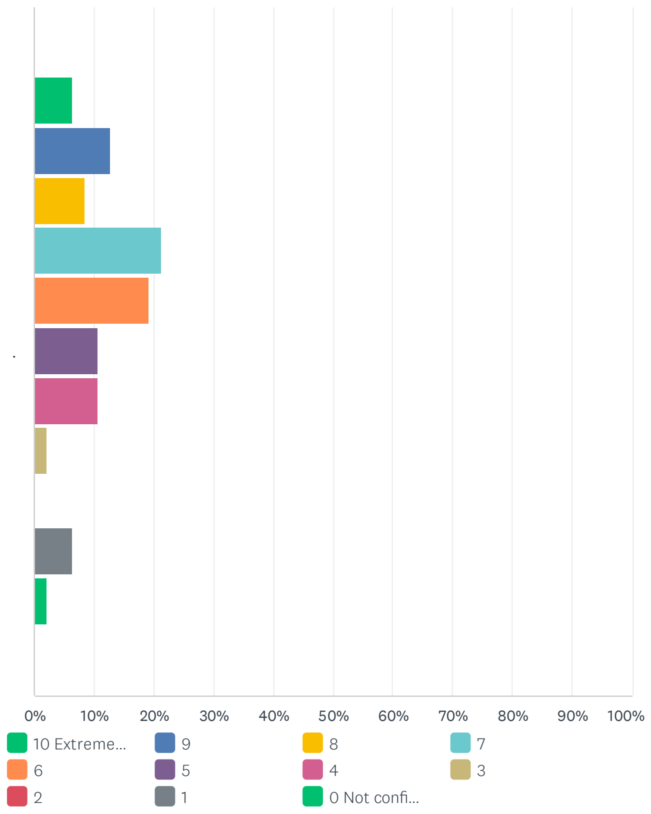
Answered: 47 Skipped: 0



	10 EXTREMELY EASY	9	8	7	6	5	4	3	2	1	0 EXTREMELY DIFFICULT	TOTA
.	4.26% 2	8.51% 4	17.02% 8	14.89% 7	14.89% 7	12.77% 6	8.51% 4	8.51% 4	6.38% 3	2.13% 1	2.13% 1	.

Q4 Using any number from 0 to 10, where 0 is not confident at all and 10 is extremely confident, what number would you use to rate how confident are you that you understand how the Cabernet program behaves?

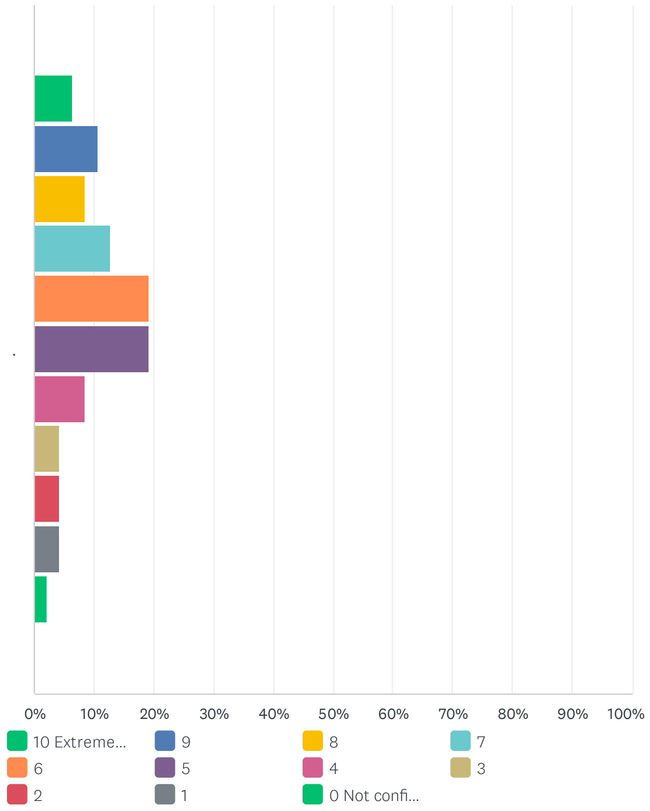
Answered: 47 Skipped: 0



	10 EXTREMELY CONFIDENT	9	8	7	6	5	4	3	2	1	0 NOT CONFIDENT AT ALL	TOT
.	6.38% 3	12.77% 6	8.51% 4	21.28% 10	19.15% 9	10.64% 5	10.64% 5	2.13% 1	0.00% 0	6.38% 3	2.13% 1	

Q5 Using any number from 0 to 10, where 0 is not confident at all and 10 is extremely confident, what number would you use to rate how confident are you that you understand how the Swift program behaves?

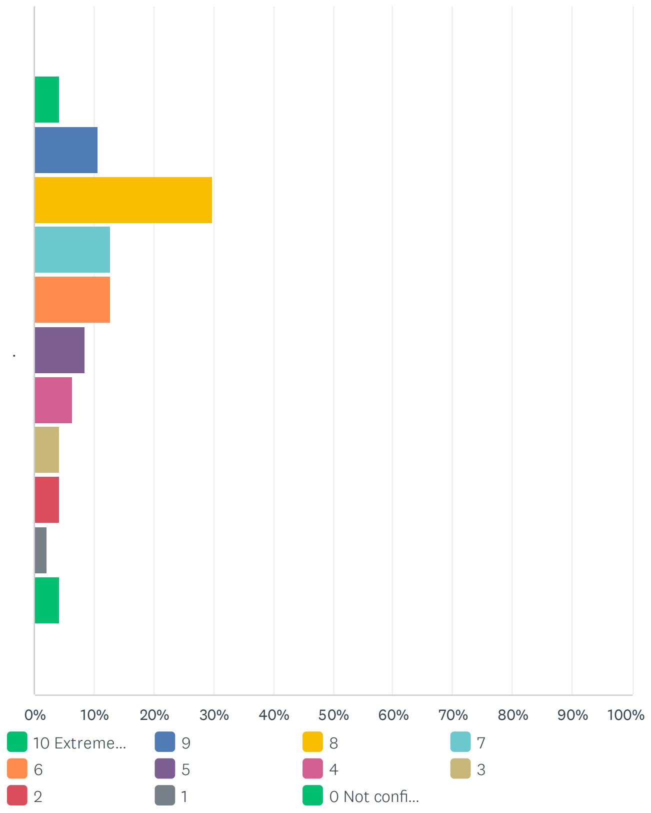
Answered: 47 Skipped: 0



	10 EXTREMELY CONFIDENT	9	8	7	6	5	4	3	2	1	0 NOT CONFIDENT AT ALL	TOTA
.	6.38% 3	10.64% 5	8.51% 4	12.77% 6	19.15% 9	19.15% 9	8.51% 4	4.26% 2	4.26% 2	4.26% 2	2.13% 1	4

Q6 Using any number from 0 to 10, where 0 is not confident at all and 10 is extremely confident, what number would you use to rate how confident are you that you understand how the SwiftUI program behaves?

Answered: 47 Skipped: 0



	10 EXTREMELY CONFIDENT	9	8	7	6	5	4	3	2	1	0 NOT CONFIDENT AT ALL	TOTA
.	4.26% 2	10.64% 5	29.79% 14	12.77% 6	12.77% 6	8.51% 4	6.38% 3	4.26% 2	4.26% 2	2.13% 1	4.26% 2	4

Q7 In the Cabernet source code what is the line number of the code that contains the formula that defines the results of the calculation?

Answered: 47 Skipped: 0

85% Answered correctly

Q8 In the Swift source code what is the line number of the code that contains the formula that defines the results of the calculation?

Answered: 47 Skipped: 0

79% Answered correctly

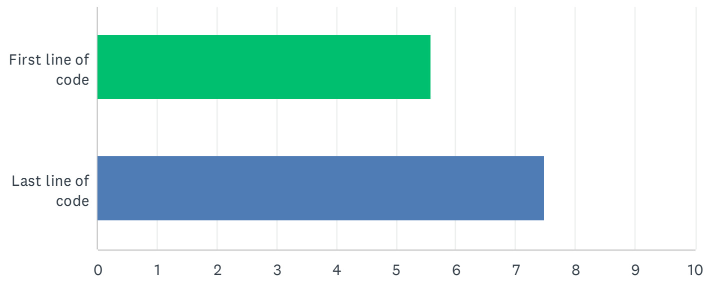
Q9 In the SwiftUI source code what is the line number of code that contains the formula that defines the results of the calculation?

Answered: 47 Skipped: 0

79% Answered correctly

Q10 At the top of the application screen there is a button containing the word "Calculate". The user taps this word to initiate the calculation operation. In the Cabernet source code which lines of code define this button?

Answered: 47 Skipped: 0

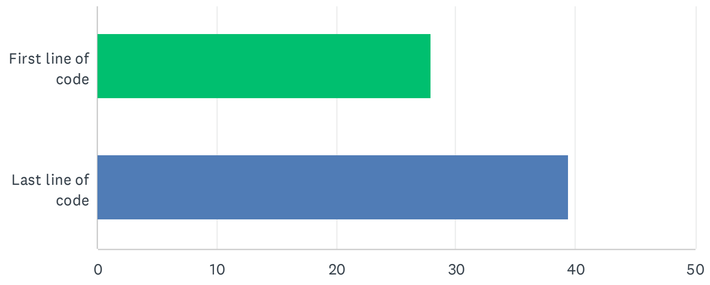


ANSWER CHOICES	AVERAGE NUMBER	TOTAL NUMBER	RESPONSES
First line of code	6	263	47
Last line of code	7	352	47
Total Respondents: 47			

81% Answered correctly

Q11 At the top of the application screen there is a button containing the word "Calculate". The user taps this word to initiate the calculation operation. In the Swift source code which lines of code define this button?

Answered: 47 Skipped: 0

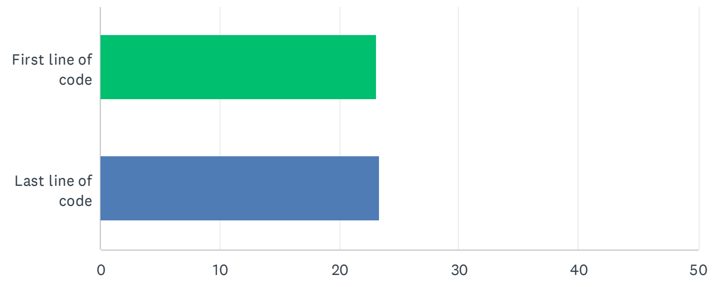


ANSWER CHOICES	AVERAGE NUMBER	TOTAL NUMBER	RESPONSES
First line of code	28	1,313	47
Last line of code	39	1,853	47
Total Respondents: 47			

66% Answered correctly

Q12 At the top of the application screen there is a button containing the word "Calculate". The user taps this word to initiate the calculation operation. In the SwiftUI source code which lines of code define this button?

Answered: 47 Skipped: 0

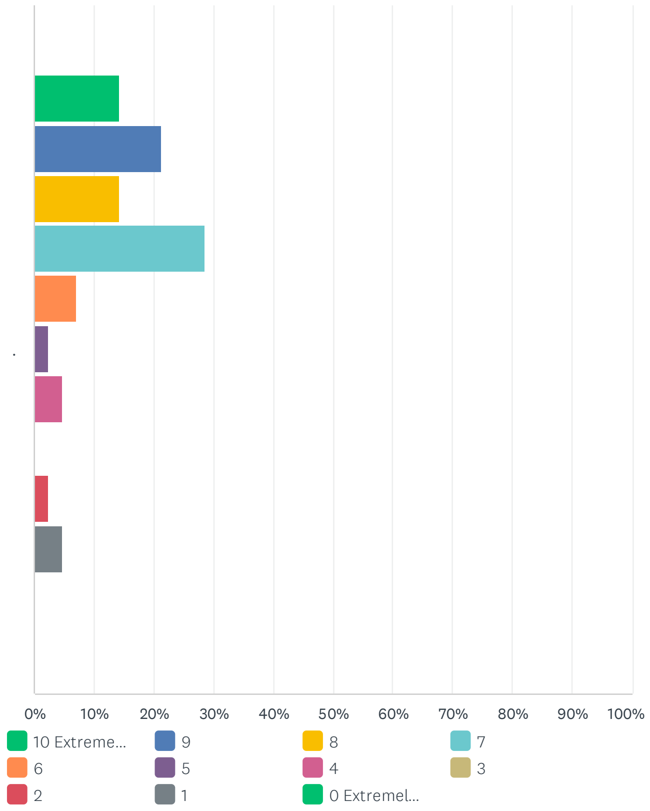


ANSWER CHOICES	AVERAGE NUMBER	TOTAL NUMBER	RESPONSES
First line of code	23	1,085	47
Last line of code	23	1,095	47
Total Respondents: 47			

66% Answered correctly

Q13 Using any number from 0 to 10, where 0 is extremely difficult and 10 is extremely easy, what number would you use to rate how easy it was for you to understand the Cabernet source code?

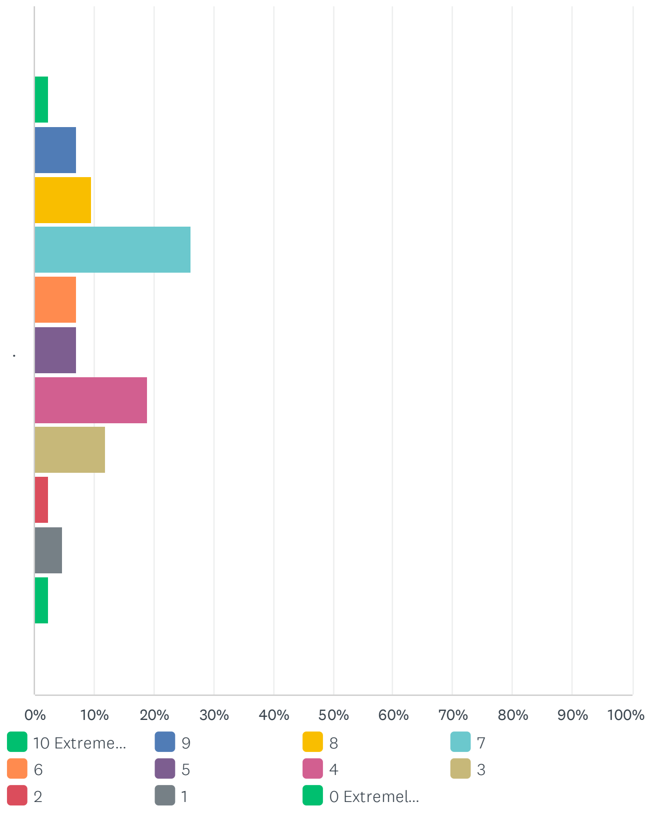
Answered: 42 Skipped: 5



	10 EXTREMELY EASY	9	8	7	6	5	4	3	2	1	0 EXTREMELY DIFFICULT	TOTAl
.	14.29% 6	21.43% 9	14.29% 6	28.57% 12	7.14% 3	2.38% 1	4.76% 2	0.00% 0	2.38% 1	4.76% 2	0.00% 0	4

Q14 Using any number from 0 to 10, where 0 is extremely difficult and 10 is extremely easy, what number would you use to rate how easy it was for you to understand the Swift source code?

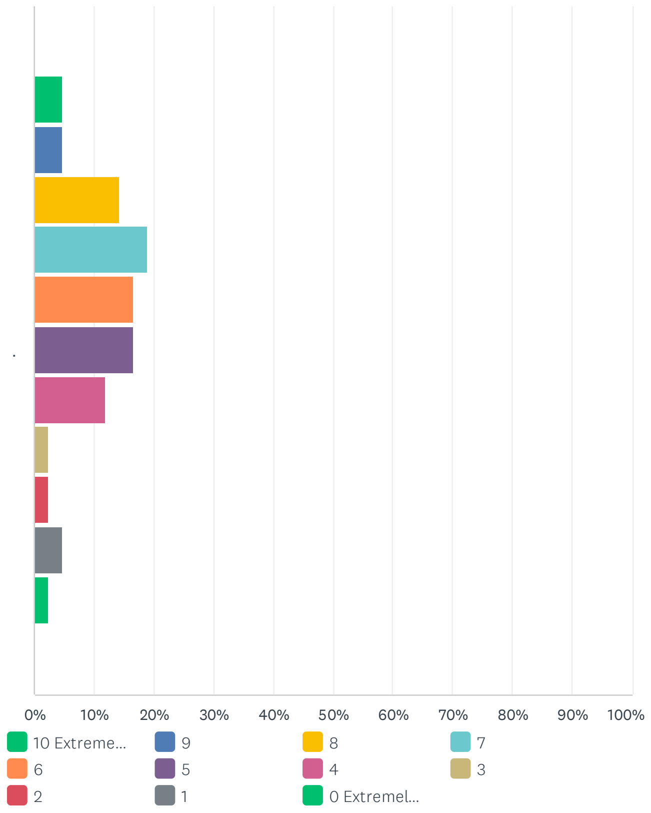
Answered: 42 Skipped: 5



	10 EXTREMELY EASY	9	8	7	6	5	4	3	2	1	0 EXTREMELY DIFFICULT	TOTAl
.	2.38% 1	7.14% 3	9.52% 4	26.19% 11	7.14% 3	7.14% 3	19.05% 8	11.90% 5	2.38% 1	4.76% 2	2.38% 1	4

Q15 Using any number from 0 to 10, where 0 is extremely difficult and 10 is extremely easy, what number would you use to rate how easy it was for you to understand the SwiftUI source code?

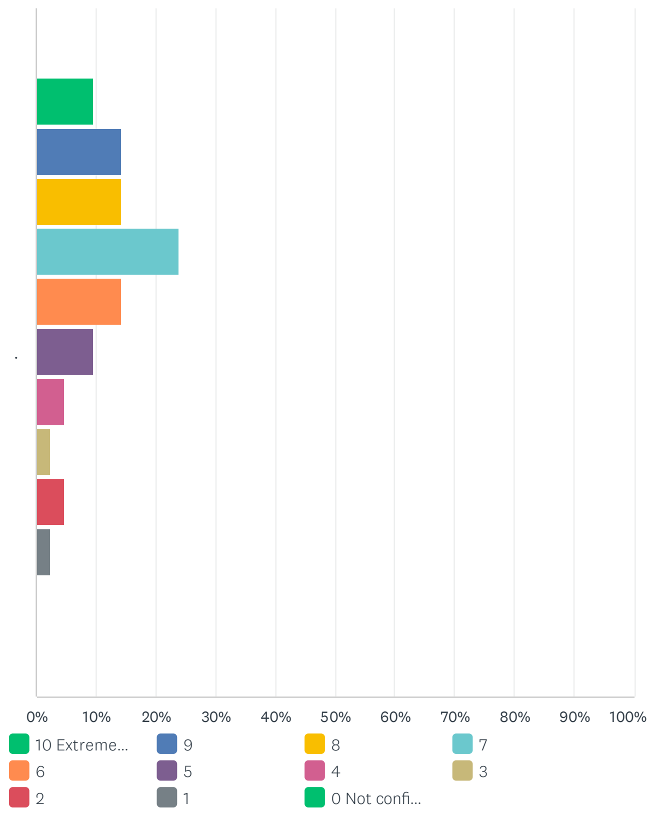
Answered: 42 Skipped: 5



	10 EXTREMELY EASY	9	8	7	6	5	4	3	2	1	0 EXTREMELY DIFFICULT	TOT
.	4.76% 2	4.76% 2	14.29% 6	19.05% 8	16.67% 7	16.67% 7	11.90% 5	2.38% 1	2.38% 1	4.76% 2	2.38% 1	

Q16 Using any number from 0 to 10, where 0 is not confident at all and 10 is extremely confident, what number would you use to rate how confident are you that you understand how the Cabernet program behaves?

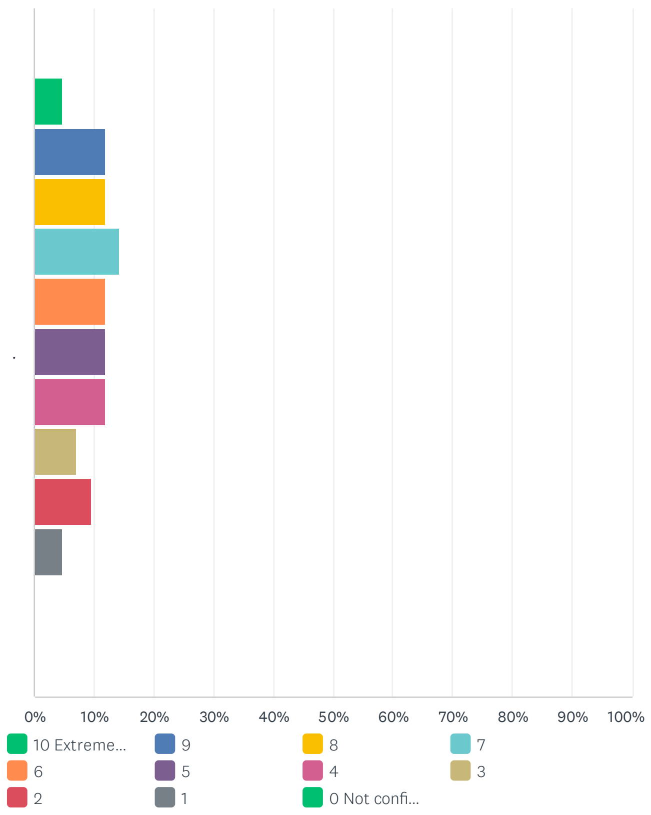
Answered: 42   Skipped: 5



	10 EXTREMELY CONFIDENT	9	8	7	6	5	4	3	2	1	0 NOT CONFIDENT AT ALL	TOTA
.	9.52% 4	14.29% 6	14.29% 6	23.81% 10	14.29% 6	9.52% 4	4.76% 2	2.38% 1	4.76% 2	2.38% 1	0.00% 0	4

Q17 Using any number from 0 to 10, where 0 is not confident at all and 10 is extremely confident, what number would you use to rate how confident are you that you understand how the Swift program behaves?

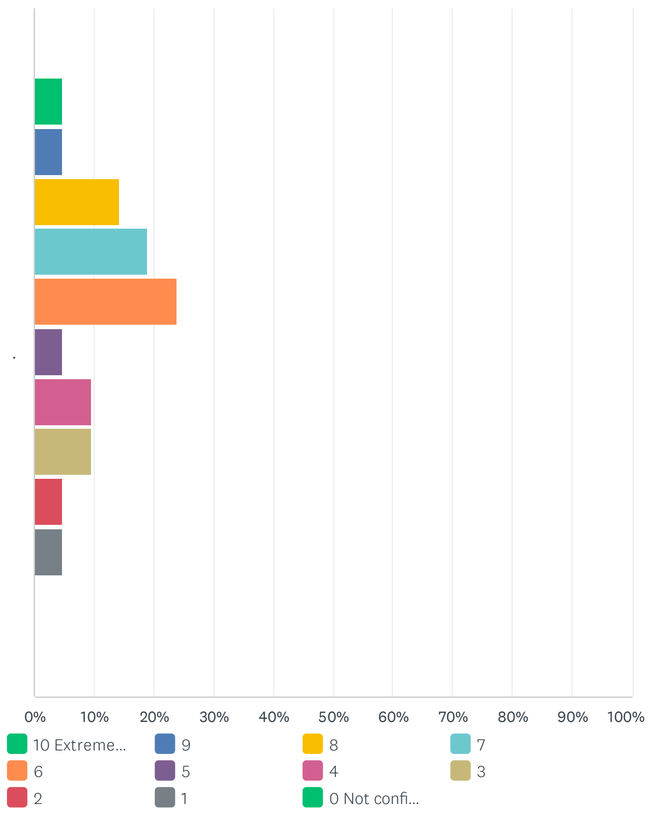
Answered: 42 Skipped: 5



	10 EXTREMELY CONFIDENT	9	8	7	6	5	4	3	2	1	0 NOT CONFIDENT AT ALL	TO
.	4.76% 2	11.90% 5	11.90% 5	14.29% 6	11.90% 5	11.90% 5	11.90% 5	7.14% 3	9.52% 4	4.76% 2	0.00% 0	

Q18 Using any number from 0 to 10, where 0 is not confident at all and 10 is extremely confident, what number would you use to rate how confident are you that you understand how the SwiftUI program behaves?

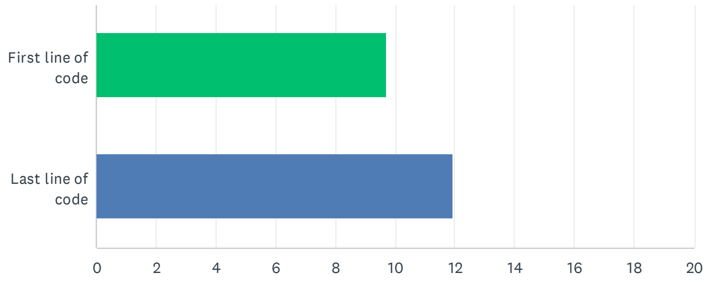
Answered: 42 Skipped: 5



	10 EXTREMELY CONFIDENT	9	8	7	6	5	4	3	2	1	0 NOT CONFIDENT AT ALL	TOTAL
.	4.76% 2	4.76% 2	14.29% 6	19.05% 8	23.81% 10	4.76% 2	9.52% 4	9.52% 4	4.76% 2	4.76% 2	0.00% 0	42

Q19 In the Cabernet source code for the real estate app which lines of code contain the instructions for launching the acreage calculator?

Answered: 42 Skipped: 5



ANSWER CHOICES	AVERAGE NUMBER	TOTAL NUMBER	RESPONSES
First line of code	10	409	42
Last line of code	12	501	42
Total Respondents: 42			

62% Answered correctly

Q20 In the Swift source code for the main real estate app which lines of code contain the instructions for launching the acreage calculator?

Answered: 42 Skipped: 5

ANSWER CHOICES	RESPONSES	
First line of code	100.00%	42
Last line of code	100.00%	42

74% Answered correctly

Q21 In the SwiftUI source code for the main real estate app which lines of code contain the instructions for launching the acreage calculator?

Answered: 42 Skipped: 5

ANSWER CHOICES	RESPONSES	
First line of code	100.00%	42
Last line of code	100.00%	42

36% Answered correctly

Q22 In the Cabernet source code for the real estate application which lines of code contain the instructions for returning to the main real estate app?

Answered: 42 Skipped: 5

ANSWER CHOICES	RESPONSES	
First line of code	100.00%	42
Last line of code	100.00%	42

57% Answered correctly

Q23 In the Swift source code for the acreage calculator which lines of code contain the instructions for returning to the main real estate app?

Answered: 41 Skipped: 6

ANSWER CHOICES	RESPONSES	
First line of code	100.00%	41
Last line of code	100.00%	41

63% Answered correctly

Q24 In the SwiftUI source code for the acreage calculator which lines of code contain the instructions for returning to the main real estate app?

Answered: 42 Skipped: 5

ANSWER CHOICES	RESPONSES	
First line of code	100.00%	42
Last line of code	100.00%	42

21% Answered correctly
Note: this question is somewhat misleading as there are no lines of code for returning to the main app

Q25 In the Cabernet source code for the revised real estate application which lines of code contain the instructions for changing the color of the owner's name field if it is left blank?

Answered: 39 Skipped: 8

ANSWER CHOICES	RESPONSES	
First line of code	100.00%	39
Last line of code	100.00%	39

87% Answered correctly

Q26 In the SwiftUI source code for the revised real estate application which lines of code contain the instructions for changing the color of the owner's name field if it is left blank?

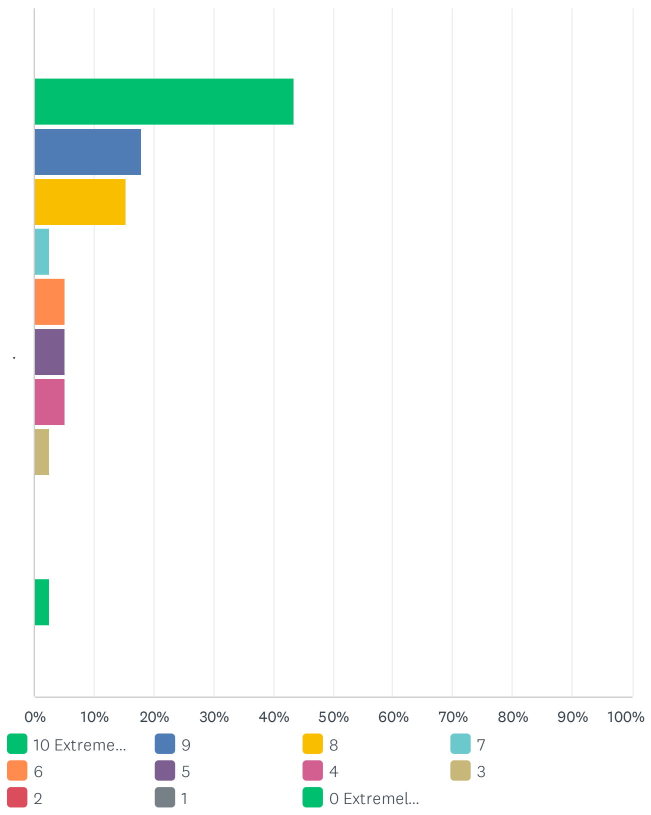
Answered: 39 Skipped: 8

ANSWER CHOICES	RESPONSES	
First line of code	100.00%	39
Last line of code	100.00%	39

62% Answered correctly

Q27 Using any number from 0 to 10, where 0 is extremely difficult and 10 is extremely easy, what number would you use to rate how easy it was for you to understand this change in the Cabernet source code?

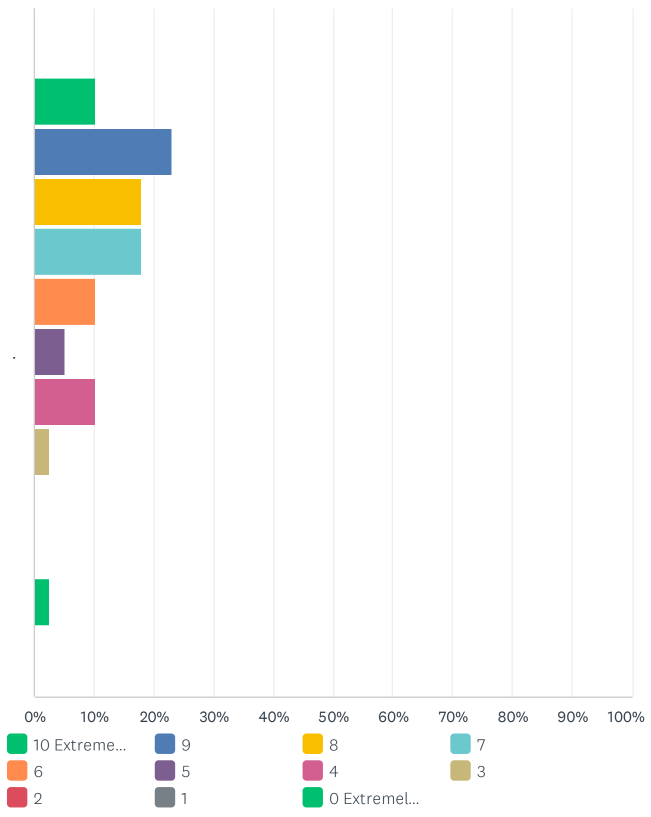
Answered: 39 Skipped: 8



	10 EXTREMELY EASY	9	8	7	6	5	4	3	2	1	0 EXTREMELY DIFFICULT	TOTAL
.	43.59% 17	17.95% 7	15.38% 6	2.56% 1	5.13% 2	5.13% 2	5.13% 2	2.56% 1	0.00% 0	0.00% 0	2.56% 1	39

Q28 Using any number from 0 to 10, where 0 is extremely difficult and 10 is extremely easy, what number would you use to rate how easy it was for you to understand this change in the SwiftUI source code?

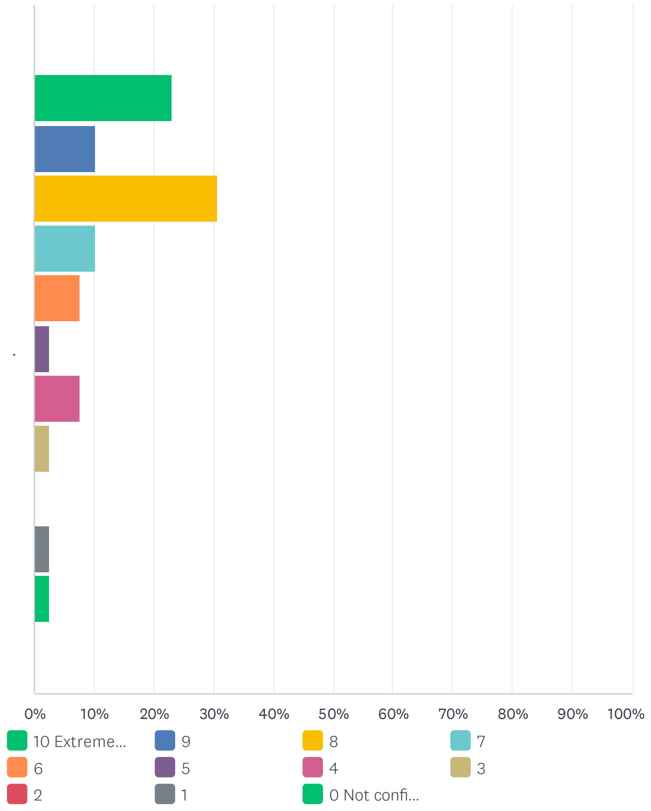
Answered: 39 Skipped: 8



	10 EXTREMELY EASY	9	8	7	6	5	4	3	2	1	0 EXTREMELY DIFFICULT	TOT
.	10.26% 4	23.08% 9	17.95% 7	17.95% 7	10.26% 4	5.13% 2	10.26% 4	2.56% 1	0.00% 0	0.00% 0	2.56% 1	

Q29 Using any number from 0 to 10, where 0 is not confident at all and 10 is extremely confident, what number would you use to rate how confident are you that you that you could add similar conditions to other fields in the Cabernet application?

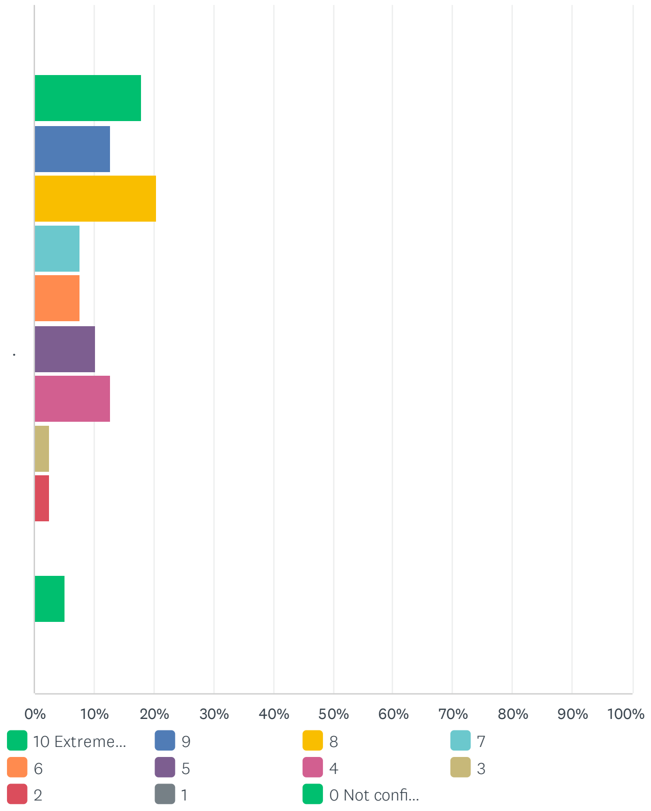
Answered: 39 Skipped: 8



	10 EXTREMELY CONFIDENT	9	8	7	6	5	4	3	2	1	0 NOT CONFIDENT AT ALL	TOTAL
.	23.08% 9	10.26% 4	30.77% 12	10.26% 4	7.69% 3	2.56% 1	7.69% 3	2.56% 1	0.00% 0	2.56% 1	2.56% 1	39

Q30 Using any number from 0 to 10, where 0 is not confident at all and 10 is extremely confident, what number would you use to rate how confident are you that you that you could add similar conditions to other fields in the SwiftUI application?

Answered: 39 Skipped: 8



	10 EXTREMELY CONFIDENT	9	8	7	6	5	4	3	2	1	0 NOT CONFIDENT AT ALL	TOTA
.	17.95% 7	12.82% 5	20.51% 8	7.69% 3	7.69% 3	10.26% 4	12.82% 5	2.56% 1	2.56% 1	0.00% 0	5.13% 2	3

Q31 Do you have any general comments, observations or suggestions on the Cabernet Tool?

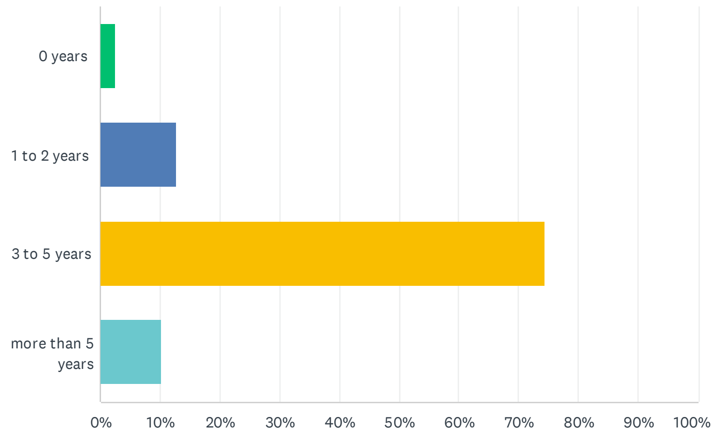
Answered: 26 Skipped: 21

Q32 How would you compare the three development tools used in these examples?

Answered: 26 Skipped: 21

Q33 How many years experience do you have in programming?

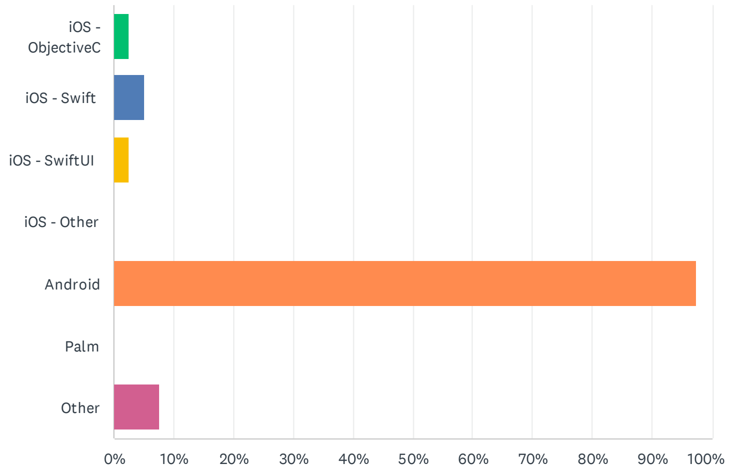
Answered: 39   Skipped: 8



ANSWER CHOICES	RESPONSES	
0 years	2.56%	1
1 to 2 years	12.82%	5
3 to 5 years	74.36%	29
more than 5 years	10.26%	4
TOTAL		39

Q34 Which of the following mobile device programming environments to you have experience in?

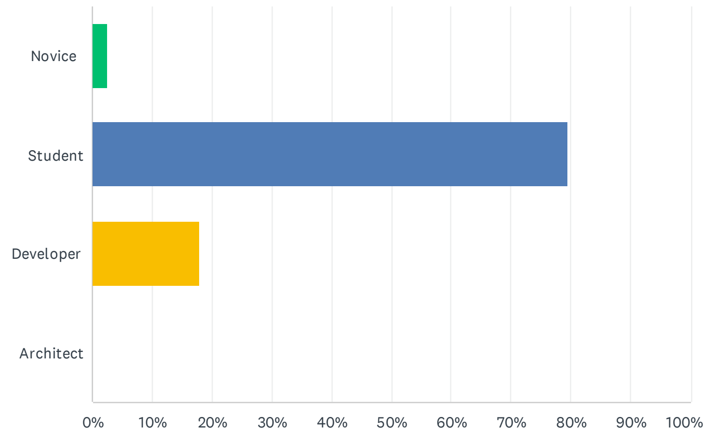
Answered: 39 Skipped: 8



ANSWER CHOICES	RESPONSES	
iOS - ObjectiveC	2.56%	1
iOS - Swift	5.13%	2
iOS - SwiftUI	2.56%	1
iOS - Other	0.00%	0
Android	97.44%	38
Palm	0.00%	0
Other	7.69%	3
Total Respondents: 39		

Q35 How would you describe your level of programming ability?

Answered: 39 Skipped: 8



ANSWER CHOICES	RESPONSES	
Novice	2.56%	1
Student	79.49%	31
Developer	17.95%	7
Architect	0.00%	0
TOTAL		39

Appendix D

Survey Respondent Comments

Q31 Do you have any general comments, observations or suggestions on the Cabernet Tool?

Answered: 26 Skipped: 21

#	RESPONSES	DATE
1	It looks great.	12/7/2022 1:55 PM
2	The Cabarnet Tool greatly simplifies much of the UI components and is able to create a similar view at a much faster rate based on the examples. However it seems to be slightly more convoluted in terms of logic and arithmetic operations, though the syntax is still basic and should probably not be particularly difficult to get used to. I do also wonder how the code will be presented for significantly larger apps that use more mobile features and libraries, or if this could just be used as a tool to go alongside Swift/SwiftUI.	12/7/2022 12:55 PM
3	For the question where it asks how it behaves, I personally think it is easier to understand this when u see each methods implementation. Hard to understand how Cabernet tool is implemented but easy to see how it "behaves"	12/6/2022 3:49 PM
4	It is much more readable in terms of figuring out what it is doing, and judging what the result will look like. However, it seems harder for if I wanted to make something specific, because I wouldn't know where to start with getting the right syntax.	12/3/2022 10:21 PM
5	It seems like it would be a pain to understand how to use fully. It can make extremely basic designs but I fail to understand how to really use it to simplify coding. I actually prefer the SwiftUI because it's strictly typed.	11/29/2022 3:21 PM
6	very different to how someone would typically code something	11/29/2022 12:54 PM
7	It is a good tool for logic.	11/25/2022 10:10 AM
8	seems very intuitive and concise, but sometimes it makes it hard to grasp what is happening for someone not familiar with coding	11/23/2022 2:42 PM
9	I feel comfused when I read Cabernet code. It's too ambigus.	11/17/2022 2:02 PM
10	It is extremely easy to understand. It is similar to writing in English.	11/11/2022 11:11 PM
11	It reminds me of markdown. It feels like it would take a little bit to get a handle on it for someone with programming experience, since it doesn't feel like a "traditional" programming language and could have some weird syntax stuff you'd need to look at a reference sheet to understand, but it also feels like it should be very easy to pick up and use for someone without programming experience.	11/11/2022 6:25 PM
12	It seems like a great tool for allowing people who aren't professional programmers to design their own apps. I am not familiar with Apple development, but it seems like a helpful tool. I would assume that this tool couldn't make as granular of detailed changes as you could make in Swift though.	11/10/2022 11:28 AM
13	I don't feel like I completely understand the logic of Cabernet, it looks like I'm calling methods but I can't see the methods so I don't know the logic behind them	11/9/2022 12:09 PM
14	No. Thank you.	11/6/2022 4:45 PM
15	I think it's good if at a glance you need to tell what a program is doing but I worry how it would perform with more complicated app interfaces. I also worry that it doesn't display enough information, but some of that might be because I have no training with Cabernet.	11/3/2022 6:58 PM
16	Cabernet seems good for custom calculators and CRUD apps. There might already be functionality for it, but directions that could change the UI to include images and other visual effects would be helpful.	11/3/2022 6:43 PM
17	it felt a bit too encapsulated, I didn't feel like I was in control, I would need time to trust it does what I want	11/3/2022 1:01 PM

18	The syntax may be confusing to people reading it for the first time.	11/3/2022 11:09 AM
19	When I entered "0" for first and last lines, this was to indicate that I did not know.	11/2/2022 4:15 PM
20	I understand simplicity is the main goal but I found it sometimes detrimental to understanding exactly what is going to happen. I'm sure good documentation will help with that.	11/2/2022 3:37 PM
21	Having scope (I assume) denoted by the number of #s makes it difficult to "skim over", since # is a rather busy character and multiple of them in succession tend to blend together.	11/2/2022 3:27 PM
22	In example #1 in the Swift source code, I find the processEachPayThis() function logic somewhat confusing. This may be due to me not actually knowing how to write in Swift, but I think maybe there could be some improvement in how that logic is written.	11/2/2022 11:14 AM
23	I could not understand why the SwiftUI code in Example 2 did not have a "cancel" button like the Carbernet and Swift implementations. I didn't know where the code actually specified how to return the the Home page. Additionally, I am assuming each of the languages build its own version of the app? As in, they aren't working together much like Java, Kotlin, and XML are used together to make an Android app?	11/2/2022 9:58 AM
24	It is very simple and easy to understand.	11/2/2022 8:50 AM
25	I didnt know how to return to main page from the Acre calculator and it wouldnt let me leave it blank so I just put 1, I originally would have said the cancel button but it looked like that was just for erasing what entries you had put in already for the calculator.	10/6/2022 2:23 PM
26	I think this would be a good tool for quick form or mockup creation, but there are many things I wonder about it. Like for these examples - can I change size of entry boxes? Can I move fields on the screen? How would a more complicated function look? I am intrigued, but scared since so much of the "brains" dictating things is hidden.	9/27/2022 12:57 PM

Q32 How would you compare the three development tools used in these examples?

Answered: 26 Skipped: 21

#	RESPONSES	DATE
1	Cabernet: Similar to Markdown Swift: Your basic average Kotlin or Java syntax SwiftUI: HTML-style embedding	12/7/2022 8:28 PM
2	The first one is the easiest one for me.	12/7/2022 1:55 PM
3	Swift is currently the most popularly used language among the three, and as a result has the deepest selection of methods and libraries that can be accessed to go very in-depth with developing an app. The SwiftUI seeks to build upon this and make it slightly easier to build a view for a fragment, though it looks to have less customization capabilities. Cabarnet creates views with significantly less lines of code than either of the former languages, though based on the examples it has not shown how it will stack up for more advanced apps. There is also some concern with how it will handle more complex logic and how easy it may be to trace the code with the monotnous way it is written.	12/7/2022 12:55 PM
4	I compared certain sections of code to one another	12/6/2022 3:49 PM
5	It seems like the Swift code was the least abstract and the Cabernet was the most abstract, with SwiftUI in between. I think the more abstract code types were easier to read to figure out how they worked, but if I was implementing an app myself the less abstract code types would ironically be easier. The more readable and easily understandable Cabernet code seems like it comes at the cost of not knowing how I would generate it myself (Unless, I guess, the point is that you don't need the syntax just right, and it can interpret what I say with NLP or something).	12/3/2022 10:21 PM
6	Cabernet seems cool and easy	12/1/2022 8:56 AM
7	I could not see the cancel button being created at all in the SwiftUI. Perhaps the picture is missing something?	11/29/2022 3:21 PM
8	In terms of how much control I feel I have over what I'm creating, Swift seems to give me the most control, then SwiftUI and then Cabernet feels very limiting for what I can create.	11/29/2022 12:54 PM
9	I feel swift code and swiftUI code is easier to understand than cabernet, however, cabernet is lightweight and fast to have something done in terms of logic and structures.	11/25/2022 10:10 AM
10	Cabernet is very concise whereas other two are more detailed as for which components are used and how they are used.	11/23/2022 2:42 PM
11	I will compare them based on more complex interaction behavior.	11/17/2022 2:02 PM
12	Cabernet doesn't feel like a programming tool, more so something like xml or markdown that gets interpreted into a style. It's much more readable than Swift or Swift UI, which could really benefit from some reasonably-named variables and comments.	11/11/2022 6:25 PM
13	Swift and SwiftUI look like traditional development languages, while Cabernet looks like a markdown file. As a programmer, I am more comfortable with the look of Swift and SwiftUI even if I don't know those languages. To me, Cabernet looks like cheating, and I don't trust it to give me the full, correct result.	11/10/2022 11:28 AM
14	Swift and SwiftUI are relatively similar, SwiftUI is much easier to understand than Swift. In order of easy to understand to hard to understand, I would rank them SwiftUI, Cabernet, Swift	11/9/2022 12:09 PM
15	The Cabernet Model is definitely much easier to write (more concise and seems more efficient.)	11/6/2022 4:45 PM
16	SwiftUI makes the most sense to me and is a good blend of simplifying the language but still giving enough information, Swift was a bit confusing to figure out what some of the code was	11/3/2022 6:58 PM

	doing and Cabernet felt too simple. SwiftUI felt like a good mix of Swift and Cabernet.	
17	Cabernet was the easiest to understand. I'm sure Swift/UI would be easier to parse if I went through tutorials / a class on iOS development, but with my current knowledge, it was definitely harder to go through than Cabernet.	11/3/2022 6:43 PM
18	the variable naming for swift and swiftUI felt mean	11/3/2022 1:01 PM
19	Cabernet is good for a quick solution. The other two are good if you want more specific options and to understand the development tools.	11/3/2022 11:09 AM
20	Cabernet seems much simpler.	11/2/2022 4:15 PM
21	I have no previous experience with any of these tools and I would say that SwiftUI was my favorite. Although Cabernet is simple and perhaps easier to understand for a beginner, as a CS student with almost 4 years of experience, I actually preferred the more complex SwiftUI because there are more details in the code to help me understand what exactly is going on and what I can choose to customize.	11/2/2022 3:37 PM
22	Cabernet, while generally easier to parse, only felt so because the code was short and entry fields were labeled by the information they store (rather than just being named "label1", "label2", etc.) Swift felt the most similar to Kotlin in syntax and semantics. SwiftUI was the easiest to understand coming from front-end development experience with JavaScript.	11/2/2022 3:27 PM
23	Swift and SwiftUI make a lot of sense, since their logic and structure are similar to Kotlin's in Android. Cabarnet reminds me a bit of a Prolog, and I'm unsure of the underlying code that executes the logic in Cabarnet. Is it meant to be used in conjunction with Swift or SwiftUI?	11/2/2022 9:58 AM
24	Cabernet easiest, Swift not too bad	11/2/2022 8:50 AM
25	swift seems more complex.	10/6/2022 2:23 PM
26	*note: enteres 61/61 on question 24 because I didn't know. I have not used SWIFT UI or SWIFT before, so all 3 were new to me. I find SWIFT most comfortable, simply because I can see more details telling me what to expect about what will happen when program runs. Cabernet seems simple, but a lot of logic is hidden, so I really wouldn't know what to expect when running, so it makes me nervous.	9/27/2022 12:57 PM

Bibliography

[1] C Russell Ed Phelps. “Proceedings of a Conference on a National Information System in the Mathematical Sciences (Harrison House, Glen Cove, New York, January 18-20, 1970).” In: (1970).

[2] Gail C. Murphy. “Beyond Integrated Development Environments: Adding Context to Software Development”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* 00 (2019), pp. 73–76. DOI: `10.1109/icse-nier.2019.00027`.

[3] John Markoff. *Machines of Loving Grace*. The Quest for Common Ground Between Humans and Robots. HarperCollins, Aug. 2015.

[4] Geert Heyman et al. “Natural Language-Guided Programming”. In: *arXiv* (2021). DOI: `10.48550/arxiv.2108.05198`. eprint: `2108.05198`.

[5] Aman Kumar and Priyanka Sharma. “Open AI Codex: An Inevitable Future?” In: *International Journal for Research in Applied Science and Engineering Technology* 11.2 (2023), pp. 539–543. DOI: `10.22214/ijraset.2023.49048`.

[6] Arghavan Moradi Dakhel et al. “GitHub Copilot AI pair programmer: Asset or Liability?” In: *arXiv* (2022). DOI: `10.48550/arxiv.2206.15331`. eprint: `2206.15331`.

[7] Dominik Sobania, Martin Briesch, and Franz Rothlauf. “Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of

GitHub Copilot and Genetic Programming”. In: *arXiv* (2021). DOI: `10.48550/arxiv.2111.07875`. eprint: `2111.07875`.

[8] Shane McIntosh et al. “Assessing the quality of GitHub copilot’s code generation”. In: *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering* (2022), pp. 62–71. DOI: `10.1145/3558489.3559072`.

[9] David Lo et al. “An Empirical Evaluation of GitHub Copilot’s Code Suggestions”. In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)* 00 (2022), pp. 1–5. DOI: `10.1145/3524842.3528470`.

[10] Hudson Heavy Industries. *Hacking With Swift*. urlhttps://www.hackingwithswift.com/100. Accessed: 2023-3-28. 2022.

[11] Jean-Baptiste Döderlein et al. “Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic?” In: *arXiv* (2022). DOI: `10.48550/arxiv.2210.14699`. eprint: `2210.14699`.

[12] Bruce W Ballard and Alan W Biermann. “Programming in Natural Language: "NLC" as a Prototype”. In: *Proceedings of the Annual Conference, ACM*. ACM. New York, NY, 1979, pp. 228–237.

[13] Alan W Biermann, Bruce W Ballard, and Anne H Sigmon. “An Experimental Study of Natural Language Programming.” In: *International Journal of Man-Machine Studies* 18.1 (1983), pp. 71–87.

[14] D E Knuth. “Literate Programming”. In: *The Computer Journal* 27.2 (Jan. 1984), pp. 97–111.

[15] D Price et al. “NaturalJava: A Natural Language Interfaxce for Programming in Java”. In: *Proceedings of the 5th ...* 2000.

[16] Michael D Ernst. “Natural Language is a Programming Language - Applying Natural Language Processing to Software Development.” In: *SNAPL* (2017).

[17] Reyes Juárez-Ramírez, Carlos Huertas, and Sergio Inzunza. “Automated Generation of User-Interface Prototypes Based on Controlled Natural Language Description.” In: *COMPSAC Workshops* (2014).

[18] S P Overmyer et al. “Conceptual modeling through linguistic analysis using LIDA”. In: *ICSE '01 Proceedings of the 23rd International Conference on Software Engineering* (May 2001).

[19] Mathias Landhaeusser and Ronny Hug. “Text Understanding for Programming in Natural Language - Control Structuresz”. In: *RAISE@ICSE* (2015), pp. 7–12.

[20] Tobias Kuhn. “A Survey and Classification of Controlled Natural Languages”. In: *arXiv.org* 1 (July 2015), pp. 121–170. arXiv: `1507.01701v1 [40]`.

[21] Attempto Project. *Attempto Project*. urlhttp://attempto.ifi.uzh.ch/site/. Accessed: 2023-3-5. 2013.

[22] Kaarel Kaljurand and Tobias Kuhn. “A Multilingual Semantic Wiki Based on Attempto Controlled English and Grammatical Framework.” In: *ESWC* 7882.Chapter 29 (2013), pp. 427–441.

[23] Norbert E Fuchs. “Reasoning in Attempto Controlled English - Non-monotonicity.” In: *CNL* 9767.Chapter 2 (2016), pp. 13–24.

[24] Rolf Schwitter et al. “A comparison of three controlled natural languages for OWL 1.1”. In: *OWLED 2008* (Apr. 2008).

[25] Iaakov Exman and Olesya Shapira. “Fast and Reliable Software Translation of Programming Languages to Natural Language.” In: *SKY* (2016), pp. 57–64.

[26] Emdad Khan and Emdad Khan. "Machine Learning Algorithms for Natural Language Semantics and Cognitive Computing". In: *2016 International Conference on Computational Science and Computational Intelligence (CSCI)* (2016), pp. 1146–1151. DOI: `10.1109/csci.2016.0217`.

[27] K Beck et al. "The agile manifesto". In: (2001).

[28] Agung Fatwanto. "Specifying translatable software requirements using constrained natural language". In: *2012 7th International Conference on Computer Science & Education (ICCSE 2012)*. IEEE, 2012, pp. 1047–1052.

[29] L Williams, E M Maximilien, and M Vouk. "Test-driven development as a defect-reduction practice". In: *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on* (2003), pp. 34–45.

[30] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, Oct. 1999.

[31] Kent Beck. *Test-driven Development*. By Example. Addison-Wesley Professional, 2003.

[32] Dan North. *Introducing BDD*. Accessed: 2023-2-28. Mar. 2006. URL: `https://dannorth.net/introducing-bdd/`.

[33] Behaviour-Driven.org. *Behaviour Driven Software*. `http://behaviour-driven.org`. Accessed: 2023-2-28. 2016.

[34] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[35] cucumber ltd. *Cucumber*. `https://cucumber.io`. Accessed: 2023-2-28. 2018.

[36] jbehave.org. *What is jBehave?* `https://jbehave.org`. Accessed: 2023-2-28. 2017.

[37] TIOBE Software. *TIOBE Index for December 2022*. `https://www.tiobe.com/tiobe-index//`. [Online; accessed 2022-12-29]. 2022.

[38] Stack Overflow. *Stack Overflow Developer Survey 2022*. `https://survey.stackoverflow.co/2022/`. [Online; accessed 2022-12-29]. 2022.

[39] Steve Ferg. *Python and Java: A Side-by-Side Comparison*. `http://pythonconquerstheuniverse.wordpress.com/2009/10/03/python-java-a-side-by-side-comparison/`. [Online; accessed 2023-2-28]. Oct. 2011.

[40] P Louridas. "Static code analysis". In: *IEEE SOFTWARE* (2006).

[41] Stefan Wagner et al. *An Evaluation of Two Bug Pattern Tools for Java*. IEEE, 2008.

[42] Microsoft. *Visual Studio*. `https://visualstudio.microsoft.com`. Accessed: 2023-3-5. 2023.

[43] Eclipse Foundation. *The Community for Open Innovation and Collaboration*. `http://www.eclipse.org`. Accessed: 2023-3-5. 2023.

[44] Apache 17. *Apache NetBeans: Fits the Pieces Together*. `https://netbeans.org`. Accessed: 2023-3-5. 2022.

[45] JetBrains SRO. *IntelliJ IDEA – the Leading Java and Kotlin IDE*. `https://www.jetbrains.com/idea/`. Accessed: 2019-4-22. 2023.

[46] JetBrains SRO. *pyCharm: The Python IDE for Professional Developers*. `https://www.jetbrains.com/pycharm/`. Accessed: 2023-3-5. 2023.

[47] Apple, Inc. *Xcode 14*. `https://developer.apple.com/xcode/`. Accessed: 2023-3-5. 2023.

[48] Tanya René Beelders and Jean-Pierre du Plessis. "The Influence of Syntax Highlighting on Scanning and Reading Behaviour for Source Code." In: *SAICSIT* (2016).

[49] Jian Li et al. "Code Completion with Neural Attention and Pointer Networks." In: *CoRR* cs.CL (2017).

[50] Robert W Sebesta. *Concepts of Programming Languages*. Addison-Wesley, Feb. 2015.

[51] Jayant Varma. *SwiftUI for Absolute Beginners*. Program Controls and Views for iPhone, iPad, and Mac Apps. Apress, Nov. 2019.

[52] Chris Barker. *Learn SwiftUI*. An introductory guide to creating intuitive cross-platform user interfaces using Swift 5. Packt Publishing Ltd, Apr. 2020.

[53] Wallace Wang. "Designing a User Interface with Constraints". In: *macOS Programming for Absolute Beginners: Developing Apps Using Swift and Xcode*. Berkeley, CA: Apress, 2017, pp. 461–480. ISBN: 978-1-4842-2662-9. DOI: 10.1007/978-1-4842-2662-9_20. URL: https://doi.org/10.1007/978-1-4842-2662-9_20.

[54] S Leonard. "The text/markdown Media Type". In: (2016).

[55] S Leonard. "Guidance on markdown: Design philosophies, stability strategies, and select registrations". In: (2016).

[56] C Tomer. *Lightweight Markup Languages*. 2015.

[57] Gruber, John. *Introducing Markdown*. https://daringfireball.net/2004/03/introducing_markdown. Accessed: 2022-12-29. 2004.

[58] Gruber, John. *Dive Into Markdown*. https://daringfireball.net/2004/03/dive_into_markdown. Accessed: 2022-12-29. 2004.

[59] Xiaoping Jia and Howard Dittmer. "Anomaly detection in dynamic programming languages through heuristics based type inference". In: *2017 Computing Conference*. IEEE. 2017, pp. 286–293.

[60] Howard Dittmer and Xiaoping Jia. "Code Generation Based on Inference and Controlled Natural Language Input". In: *Computer Science & Information Technology (CS & IT) 2020* (2020), pp. 89–101. DOI: 10.5121/csit.2020.100408.

[61] Howard Dittmer and Xiaoping Jia. "Programmer Productivity Enhancement Through Controlled Natural Language Input". In: *International Journal of Software Engineering & Applications* 11.3 (2020), pp. 1–18. ISSN: 0976-2221. DOI: 10.5121/ijsea.2020.11301.

[62] Lutz Prechelt. "Rethinking Productivity in Software Engineering". In: (2019), pp. 3–11. DOI: 10.1007/978-1-4842-4221-6_1.

[63] L Prechelt. "An empirical comparison of seven programming languages". In: *Computer* 33.10 (2020), pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/2.876288.

[64] William R Nichols. "The End to the Myth of Individual Programmer Productivity". In: *IEEE Software* 36.5 (2019), pp. 71–75. ISSN: 0740-7459. DOI: 10.1109/ms.2019.2908576.

[65] Yannis Smaragdakis. "Next-Paradigm Programming Languages: What Will They Look Like and What Changes Will They Bring?" In: (2019). eprint: 1905.00402.

[66] John F. Pane, Chotirat "Ann" Ratanamahatana, and Brad A. Myers. "Studying the language and structure in non-programmers' solutions to programming problems". In: *International Journal of Human-Computer Studies* 54.2 (2001), pp. 237–264. ISSN: 1071-5819. DOI: 10.1006/ijhc.2000.0410.

[67] Toby Jia-Jun Li et al. "Interactive Task and Concept Learning from Natural Language Instructions and GUI Demonstrations". In: *arXiv* (2019). eprint: 1909.00031.

[68] Judith Good and Kate Howland. "Programming language, natural language? Supporting the diverse computational activities of novice programmers." English. In: *J. Vis. Lang. Comput.* 39 (2017), pp. 78–92. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2016.10.008.

[69] Tiantian Gao. "Controlled Natural Languages and Default Reasoning". In: (2019). eprint: 1905.04422.

[70] Sharvari Nadkarni et al. "Semi natural language algorithm to programming language interpreter". In: 2016 International Conference on Advances in Human Machine Interaction (HMI. 2016, pp. 1–4. ISBN: 978-1-4673-8810-8. DOI: 10.1109/hmi.2016.7449190.

[71] Rada Mihalcea, Hugo Liu, and Henry Lieberman. "Computational Linguistics and Intelligent Text Processing, 7th International Conference, CICLing 2006, Mexico City, Mexico, February 19-25, 2006. Proceedings". English. In: vol. 3878. Computational Linguistics and Intelligent Text Processing. 2006, pp. 319–330. ISBN: 9783540322054.

[72] Sida I Wang et al. "Naturalizing a Programming Language via Interactive Learning." In: *ACL* (2017), pp. 929–938. DOI: 10.18653/v1/p17-1086.

[73] Shuhan Yan et al. "Are the Code Snippets What We Are Searching for? A Benchmark and an Empirical Study on Code Search with Natural-Language Queries". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* 00 (2020), pp. 344–354. DOI: 10.1109/saner48275.2020.9054840.

[74] Miltiadis Allamanis et al. "A Survey of Machine Learning for Big Code and Naturalness". In: *arXiv* (2017). eprint: 1709.06182.