CODE GENERATION BASED ON CONTROLLED NATURAL LANGUAGE INPUT

Howard Dittmer and Xiaoping Jia

Jarvis College of Computing and Digital Media, DePaul University, Chicago, IL

ABSTRACT

Over time the level of abstraction embodied in programming languages has continued to grow. However, most programming languages still require programmers to conform to rigid constructs. These constructs have been implemented in the name of efficiency for the computer. 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 an outline-based syntax. Using heuristics and inference to analyze and determine the programmer's intent, this tool chain can create mobile applications. Using 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, performance is maintained. In this paper, we compared sample applications created in Swift, SwiftUI, and CABERNET. The CABERNET implementations were consistently shorter than those produced in the other two languages. In addition, users surveyed consistently found CABERNET easier to understand.

KEYWORDS

Controlled Natural Language, Literate Programming, Programming Language, Computer-aided Software.

1. INTRODUCTION

Computer programming provides 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 of 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 1960's. These approaches include everything from requirements capture to the evaluation of code for potential errors. Unfortunately, most of these tools have been applied piecemeal to the problem of program development. 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 in effect 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 proponents such as Doug Engelbart and Terry Winograd [1–3]. To that end, this approach combines inference with developer interaction to create robust solutions to program needs while maximizing developer productivity.

While significant research 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, which is easy to understand and accessible for novices. This combination will provide the opportunity for significant improvements in programmer productivity and quality.

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] and GitHub CoPilot [5], an OpenAI based code generation tool. Another approach in this area is Genetic programming [6]. 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. These tools have been described as AI Pairs Programming and 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 [7] 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. But these success rates are not such that a programmer can expect a correct solution without significant review and refinement. Another study of GitHub Copilot generated code [8] found code correctness ranged from 57% on Java examples to 27% on JavaScript examples.

These results indicate that GitHub Copilot will likely be valuable tools for programmers in the future. 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 creating programs.

2.2. Controlled Natural Languages

A natural language programming language has long been a goal in the programming community. In 1983 Biermann, Ballard and Sigmon introduced NLC [9,10], a natural language notation, which was interpreted directly to an output. In 1984 Knuth proposed Literate Programming [11], which combined TEX and Pascal to produce a vocabulary that had the primary goal of documenting for humans what the programmer desires. Literate Programming makes efforts to improve the readability of programs. However, it does this by adding English content to program code. The result is a program which is more verbose than the Pascal upon which it is built. In 2000 Price, Rilofff, Zachary, and Harvey introduced Natural Java [12], a notation that allows the programmer to define a procedure in English, which is converted to Java.

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 [13] including ". . . 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 [14]. 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 [15].

In another approach, [16], 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.

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.

2.3. Requirements Capture

Requirements capture is an area where Controlled Natural Language approaches have previously been used [17]. For some years, the agile development community has sought to develop better ways to capture user requirements. Test-Driven Development (TDD) [18] was initially associated with agile development in Kent Beck’s book on eXtreme Programming [19] and then expanded upon in his book on the subject [20]. 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. More recently, parts of the agile community have embraced Behavior-driven Development (BDD) as a starting point. Behavior-driven Development [21, 22] describes the user’s requirements as a series of behaviors that can be converted into tests. These tests are then used as those envisioned in Test-driven Development. These behaviors are described in a natural language form. As such, BDD acts as a front-end for TDD. Cucumber [23, 24] and jBehave [25] are popular tools that allow developers to capture their requirements in an end-user-friendly format and produce a test suite for TDD applications. While these methodologies and associated tools enable the user to describe the requirements in a natural language format, they still require the program to 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 [26] and the StackOverflow annual programmer survey [27]. Dynamic programming languages have gained a following because they have helped improve programmers’ productivity. The combination of dynamic typing and concise syntax results in fewer lines of code required to achieve the desired result. These advantages have led to claims of productivity gains from 5 to 10 times [28]. 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 [29]. 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 declared but never used, and other errors that can be identified by reviewing the source code. In addition to stylistic checks, traditionally the approach of linters, these tools have taken more ambitious approaches, such as using bug patterns. Two of the most popular and successful products in the area are FindBug and PMD [30]. They have proved very useful in finding bugs in already-written code. They help improve the code quality but do not help create the code.

2.6. Integrated Development Environments

The Integrated Development Environment (IDE) is the most used tool for developers. Among the many capabilities a modern IDE provides is syntax highlighting [31], 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 [32], automatically completing 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 [33] 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 1.1 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.

```
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 1.1. Swift code for Simple Button

In 2019 Apple introduced SwiftUI [34], which utilizes a declarative syntax for describing the program's user interface. Declarative syntax [35] describes the results the programmer wants to achieve but not how to achieve those results. Listing 1.2 includes the SwiftUI code required to create the same button as in the previous example but does it in seven lines of code. 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.

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

Listing 1.2. SwiftUI code for Simple Button

3. OUR APPROACH

The goal of our work is to provide a highly readable, flexible, extensible, and easy-to-learn development methodology based on a CNL. To that end, we have developed CABERNET (**C**ode gener**A**tion **B**as**E**d on cont**R**olled **N**atural languag**E** inpu**T**), an approach that allows a programmer to define a computer program using a Controlled Natural Language (CNL). Figure 1 lists the key advantages of the CABERNET development approach.

3.1. 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.

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

Fig. 1. Key Characteristics of CABERNET

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. The characteristics of the proposed Controlled Natural Language model are listed in Figure 2. The result is that CABERNET programs are consistently more concise than other similar approaches such as Literate Programming.

3.2. 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 1 includes some examples of these lists of synonyms.

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 might include the word “to” as it could 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.

3.3. 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.

- 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 [36] 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

Fig. 2. Characteristics of CABERNET CNL

Figure 3 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.

Table 1. CABERNET Synonym Examples.

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"

4. Notation

4.1. Markdown

The notation for the Controlled Natural Language tool is based on the Markdown [37, 38] lightweight markup language. Markdown was created in 2004 by John Gruber [39]. 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.

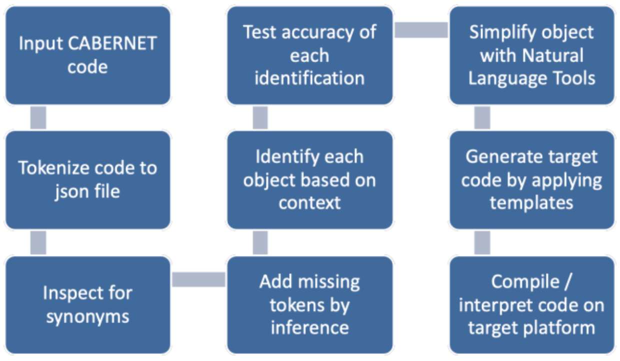

Input CABERNET code
Tokenize code to json file
Inspect for synonyms
Add missing tokens by inference
Identify each object based on context
Test accuracy of each identification
Simplify object with Natural Language Tools
Generate target code by applying templates
Compile / interpret code on target platform

Fig. 3. Processing CABERNET program

4.2. 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 4.

- #, ##, ###, 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

Fig. 4. 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 Listing 1.3. Line 1 of this code identifies the basic application. Lines 2 and 17 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.

Outline entries that start with a ‘*’ describe the content of the various objects. Entries such as 6 and 21 that start with a verb describe actions to be taken when clicking the object. Entries such as line 22 that begin with a characteristic describe the format of that field. Entries like that beginning on line 30 define a calculation that is used to populate the field. Lines like 8 and 12, which do not fall into other categories provide a default entry for the field.

```
1       # App
2       ## Scene
3           * home
4       ### “Home Listing”
5       ### “Acreage Calculator”
6           * to acreagecalc
7       ### “Owner Name”
8           * enter some text
9       ### “City”
10          * enter some text
11      ### “State”
12          * xx
13      ### “Zip Code”
14          * xxxxx-xxxx
15      ### “Active Listing”
16          * option selected
17      ## Screen
18          * acreagecalc
19      ### “Acreage Calculator”
20      ### “Calculate”
21          * calculate Lot Acreage
22          * background green
23      ### “Cancel”
24          * cancel entry
25      ### “Lot Width”
26          * enter some text
27      ### “Lot Depth”
28          * enter some text
29      ### “Lot Acreage”
30          * Divide Lot Width times Lot Depth by 43560
```

Listing 1.3. CABERNET Sample Code.

5. THESAURUS

The use of a CNL means that multiple names can describe an object in the user interface. For example, in Listing 1.3, line 2, we refer to one screen of the application as a “Scene.” On line 17, 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 1.

6. 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, line 30 in Listing 1.3 describes the calculation of the value displayed in the object.

```
Divide Lot Width times Lot Depth by 43560
```

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

Divide Lot Width times Lot Depth ***by*** 43560

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 Lot Width, and Lot Depth.

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 “divide” at the beginning of the line applies.

If the programmer had entered Width rather than Lot Width or Depth rather than Lot Depth, 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 12, and 14 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.

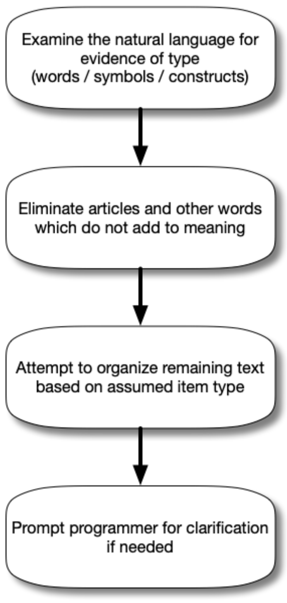

Examine the natural language for evidence of type (words / symbols / constructs)
Eliminate articles and other words which do not add to meaning
Attempt to organize remaining text based on assumed item type
Prompt programmer for clarification if needed

Fig. 5. Construct Processing

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

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.

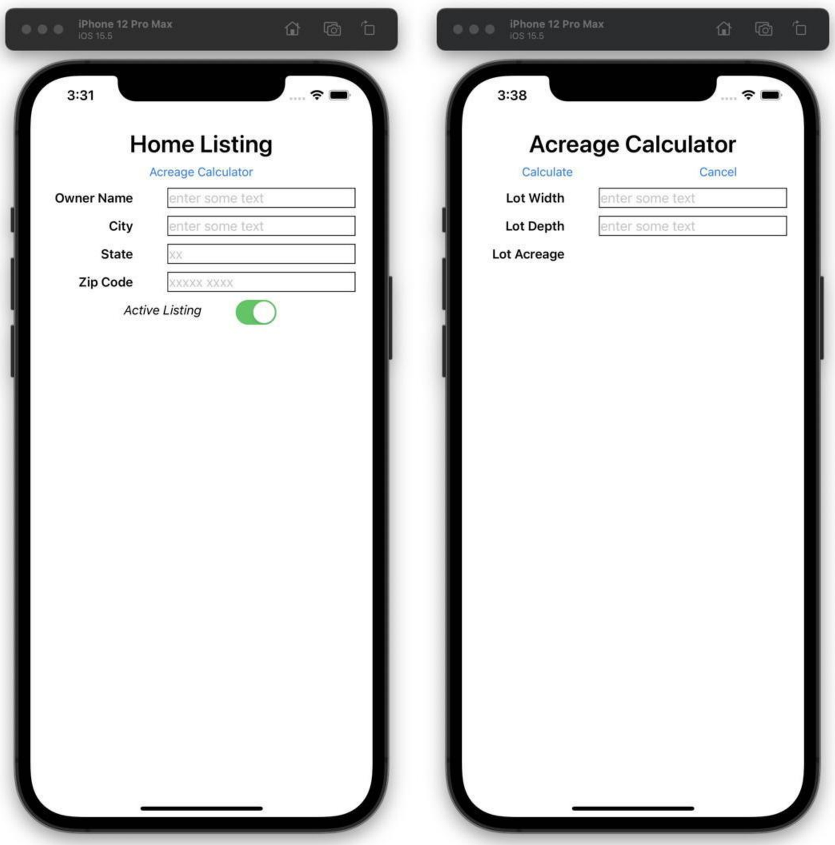
Figure 6 represents the output of our example application. In Figure 6(a), we have the entry screen for a real estate application. The App, Scene and Screen bullets are for organization and are used to separate the application by screens. “Calculate” and “Acreage Calculator” are actions and become buttons. The descriptions of the actions taken for each of the tappable objects are listed as sub-bullets. Next comes multiple blank fields for the owner’s name, city, state, and zip code. Finally, there is the options field represented by switch objects. Depending upon the platform targeted, these could alternately be check- boxes. In this case, the option field 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 6(b) is the second screen of the application and includes an acreage calculator. This screen contains two blanks filled with the lot width and lot depth. Finally, there is a calculated field representing the size of the lot in acres. As previously described, the text defines this final field in lines 29 through 30. This calculation is triggered by tapping the “Calculate” button described in lines 20 through 22.

7. Evaluation

7.1. Advantages and Limitations

Much of the approach’s power comes from the flexibility of nomenclature. This flexibility comes from the use of thesaurus, which allows 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.



(a) Real Estate App (b) Acreage Calculator

Fig. 6. App example screens.

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.

7.2. Code Size

The key process metrics that we seek to address with CABERNET include code development speed, clarity, and size. As can be seen from the example 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. To evaluate CABER- NET we have compared its code with the alternate ways of implementing some iPhone applications. The program in Listing 1.3 is one of the examples used in this comparison. While it took 30 lines of code to implement this application in CABERNET the same program took 211 and 96 lines of code in Swift and SwiftUI respectively. Note that these line counts do not include lines that include only brackets and spaces. Table 2 shows a comparison of the code required by each of the languages to create this program and the other two examples. The second example involves adding highlighting to one of the fields based on the content of the field. This adds 10 lines of code to the SwiftUI program but only one line of code to the CABERNET program. As we can see, Swift requires over seven times as many lines of code as does CABERNET to implement these screens. While the SwiftUI implementation is shorter than the Swift implementation, it still requires more than 3 times as many lines of code as does CABERNET.

Table 2. Comparison of code size

Example		**CABERNET**	**Swift**	**SwiftUI**
Real Estate App	Lines of Code	30	211	96
	Comparison with CABERNET	1X	7.3X	3.3X
Real Estate App Revised	Lines of Code	31		106
	Comparison with CABERNET	1X		3.5X
Tip Calculator	Lines of Code	14	104	57
	Comparison with CABERNET	1X	7.4X	4.1X

Much of this Swift and SwiftUI code implements things that CABERNET handles as default values and constructs. One clear example of this is the actual calculation of the acreage value. In the CABERNET version, the calculation is defined in line 30. In addition, line 21 describes the action to be taken when we tap the subject button. To perform the same calculation in Swift, we need to include 3 lines to declare the variables involved, 9 lines to create the button, and 6 lines to perform the actual calculation. That is a total of 18 lines of code. For the SwiftUI implementation, we have 3 lines for declaring the variables, 4 lines to define the button, and 4 lines for the method to perform the calculation. This is a total of 11 lines of code. Again, these line counts do not include any lines which include brackets. 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.

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.

8. 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.

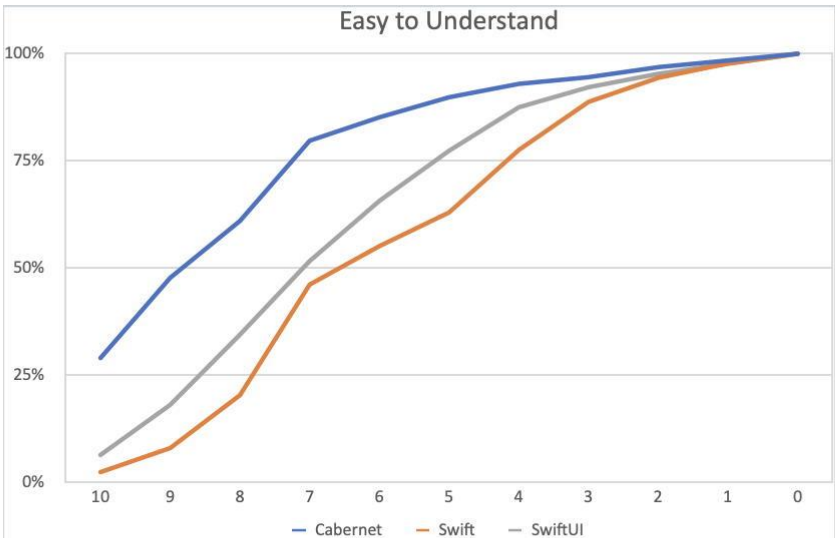


Fig. 7. Ease of Understanding

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 two major University 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. 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 7. 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 respectively.

Table 3. Mean Ease of Understanding Scores

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

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. The chart in Figure 8 shows the actual responses for CABERNET and SwiftUI. From this, you can see that while the SwiftUI results form a normal distribution around it's 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.

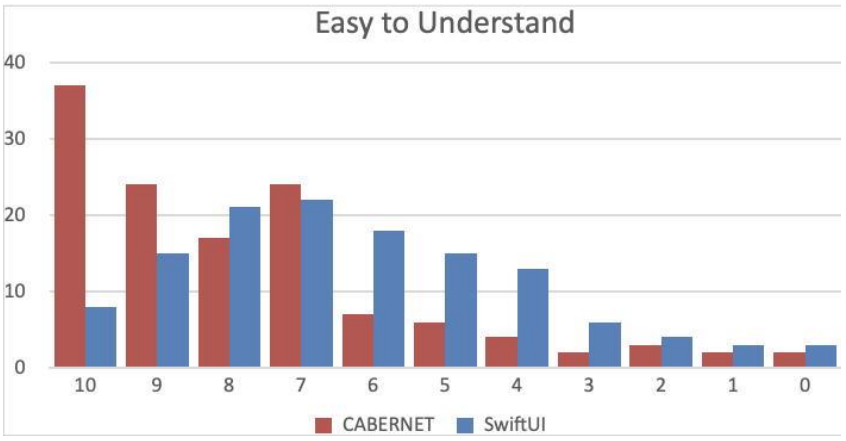


Fig. 8. CABERNET Responses vs SwiftUI

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 3.

9. Respondent Feedback

In addition to the quantitative responses based on examples, application survey respondents were offered the opportunity to provide comments about the various programming options. In total there were 52 comments submitted. Some of these responses had to do with the mechanics of the survey itself rather than the tools or were general. Twenty-one were simply positive comments about CABERNET. Seventeen 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 mock-up creation, but there are many things I wonder about it. As for these examples - can I change the size of the entry boxes? Can I move fields on the screen? How would 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.

10. RELATED WORK

10.1. 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. In his research, William Nichols [40] 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. Lutz Prechelt [41] highlights the wide variety of things that affect the program development process's overall productivity. The choice of programming language [42] 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 because of using scripting languages.

10.2. Next Paradigm Programming Languages

Yannis Smaragdakis [43] 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.

10.3. Natural Programming Languages

To date, none of the efforts to use natural language as a programming language have been accepted by mainstream programming applications. Good and Howland [44] 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. 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 [45] 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). From this research, 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.

Wang, Ginn, et al. [46] have applied the concept of a language that learns from the programmers to Natural Language input. 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. This work demonstrates how natural language programming can be effective when it grows based on user input. 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.

10.4. Code Snippets

One of the most common processes used by developers is searching online development resources to identify approaches to solving specific problems. StackOverflow is a site frequented by many programmers seeking to find answers to their programming questions. Yan et al. [47] created CosBench which 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. [48] identified a depth of work undertaking this same approach. These are interesting tools but are not programming methodologies in themselves.

11. Conclusion

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.

REFERENCES

[1] G.C.Murphy, Beyond Integrated Development Environments: Adding Context to Software Development, 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) pp. 73-76, 2019.

[2] J. Markoff, Machines of Loving Grace, The Quest for Common Ground Between Humans and Robots. Harper Collins, 2015.

[3] L. Fisher, Siri, Who is Terry Winograd. https://www.strategy-business.com/article/Siri-Who-Is-Terry- Winograd. Accessed: 2023-2-27.

[4] G. Heyman, R. Huysegems, P. Justen and T. Cutsem, Natural Language-Guided Programming, arXiv, 2021.

[5] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. Desmarais, Z. Ming, Jiang, GitHub Copilot AI pair programmer: Asset or Liability, arXiv, 2022.

[6] D. Sobania, M. Briesch and F. Rothlauf, Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of GitHub Copilot and Genetic Programming, arXiv, 2021.

[7] S. McIntosh, W. Shang, G. R. Perez, B. Yetistiren, I. Ozsoy and E. Tuzun, Assessing the quality of GitHub copilot’s code generation, Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, 2022.

[8] D. Lo, S. McIntosh, N. Novielli, N. Nguyen and S. Nadi, An Empirical Evaluation of GitHub Copilot’s Code Suggestions, 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), 2022.

[9] B. Ballard and A. Biermann, Programming in Natural Language: ”NLC” as a Prototype, Proceedings of the Annual Conference, ACM, 1979.

[10] A. Biermann, B. Ballard, and A. Sigmon, An Experimental Study of Natural Language Programming. In: International Journal of Man-Machine Studies 18.1, pp. 71–87., 1983

[11] D. Knuth, Literate Programming, The Computer Journal,pp. 97-111., 1984.

[12] D. Price, E. Rilofff, J. Zachary, and B. Harvey, NaturalJava: A Natural Language Interface for Programming in Java. In: Proceedings of the 5th . . ., 2000.

[13] M. Ernst, Natural Language is a Programming Language - Applying Natural Language Processing to Software Development., SNAPL, 2017.

[14] R. Juarez-Ramırez, C. Huertas, and S. Inzunza, Automated Generation of User-Interface Prototypes Based on Controlled Natural Language Description., COMPSAC Workshops, 2014

[15] S. Overmyer, and B. Lavoie, Conceptual modeling through linguistic analysis using LIDA, ICSE ’01 Proceedings of the 23rd International Conference on Software Engineering, May 2001.

[16] M. Landhaeusser, and R. Hug, Text Understanding for Programming in Natural Language - Control Structures, RAISEICSE pp. 7-12, 2015.

[17] A. Fatwanto, Specifying translatable software requirements using constrained natural language, 2012 7th International Conference on Computer Science & Education (ICCSE 2012), pp. 1047-1052, 2012.

[18] L. Williams, E. Maximilien, and M. Vouk, Test-driven development as a defect-reduction practice, Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium pp. 34-45, 2003.

[19] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley Professional, October 1999.

[20] K. Beck, Test-driven Development, Addison-Wesley Professional, 2003.

[21] D. North, Introducing BDD, https://dannorth.net/introducing-bdd/, Accessed: 2023-2-28, 2006.

[22] behavior-driven.org, Behaviour Driven Software, http://behaviour-driven.org, Accessed: 2023-2-28, 2016.

[23] M. Wynne, A. Hellesoy, and S. Tooke, The cucumber book: behaviour-driven development for testers and developers, Pragmatic Bookshelf, 2017.

[24] Cucumber LTD., emphCucumber, https://cucumber.io, Accessed: 2023-2-28, 2018.

[25] jbehave.org, What is jBehave?, https://jbehave.org, Accessed: 2023-2-28, 2017.

[26] TIOBE Software, TIOBE Index for December 2022, https://www.tiobe.com/tiobe-index/, Accessed: 2022-12-29, 2022.

[27] Stack Overflow, Stack Overflow Developer Survey 2022, https://survey.stackoverflow.co/2022/, Accessed: 2022-12-29, 2022.

[28] S.Ferg, Python and Java: A Side-by-Side Comparison, http://pythonconquerstheuniverse.wordpress.com/2009/10/03/python-java-a-side-by-side-comparison/, Accessed: 2023-2-28, 2011.
[29] P. Louridas, Static code analysis, IEEE Software, 2006.
[30] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer and M. Schwalb, An Evaluation of Two Bug Pattern Tools for Java, IEEE, 2008.
[31] T. R. Beelders and J. du Plessis, The Influence of Syntax Highlighting on Scanning and Reading Behaviour for Source Code, SAICSIT, 2016.
[32] J. Li, Y. Wang, I. King and M. Lyu, Code Completion with Neural Attention and Pointer Networks, CoRR, 2017.
[33] R. Sebesta, Concepts of Programming Languages, Addison-Wesley, 2015.
[34] J. Varma, SwiftUI for Absolute Beginners, Apress, 2019.
[35] C. Barker, Learn SwiftUI, Packt Publishing Ltd, 2020.
[36] S. Leonard, The text/markdown Media Type, 2016.
[37] S. Leonard, Guidance on markdown: Design philosophies, stability strategies, and select registrations, 2016.
[38] C. Tomer, Lightweight Markup Languages, 2015.
[39] J. Gruber, Introducing Markdown, https://daringfireball.net/2004/03/introducing markdown, Accessed: 2022-12-29, 2004.
[40] W. Nichols, The End to the Myth of Individual Programmer Productivity, IEEE Software pp. 71-75, 2019.
[41] L. Prechelt, Rethinking Productivity in Software Engineering, 2019.
[42] L. Prechelt, An empirical comparison of seven programming languages, Computer pp. 23-29, 2020.
[43] Y. Smaragdakis, Next-Paradigm Programming Languages: What Will They Look Like and What Changes Will They Bring?, 2019.
[44] J. Good and K. Howland, Programming language, natural language? Supporting the diverse computational activities of novice programmers, Journal of Visual Language and Computing pp. 78-92, 2017.
[45] T. Gao, Controlled Natural Languages and Default Reasoning, 2019.
[46] S.I. Wang, S. Ginn, P.Liang and C.D. Manning, Naturalizinga Programming Language via Interactive Learning, ACL pp. 929-938, 2017.
[47] S. Yan, H. Yu, Y. Chen, B. Shen and L. Jiang, Are the Code Snippets What We Are Searching for? A Benchmark and an Empirical Study on Code Search with Natural-Language Queries, 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) pp. 344-354, 2020.
[48] M. Allamanis, E. T. Barr, P. Devanbu and C. Sutton, A Survey of Machine Learning for Big Code and Naturalness, arXiv, 2017.

AUTHORS

Howard Dittmer received his MS in Software Engineering from DePaul University, and he received a BS in mechanical engineering from Virginia Tech. Currently, he is pursuing his PhD in Computer Science from DePaul University. His research interests include Software Engineering.

Xiaoping Jia received his undergraduate degree and Master’s degree in Computer Science from Fudan University, Shanghai, China. He received his Ph.D. in Computer Science from Northwestern University. He is currently the Director of Institute for Software Engineering at DePaul University. His research interests include Software Engineering, Systems Development, Programming Languages.