Programmer Productivity Enhancement Through Controlled Natural Language Input

Howard Dittmer and Xiaoping Jia

School of Computing, College of Computing and Digital Media, DePaul University, Chicago, Illinois, U.S.A.

Abstract. We have created **CABERNET**, a Controlled Nature Language (CNL) based approach to program creation. **CABERNET** allows programmers to use a simple outline-based syntax. This allows increased programmer efficiency and syntax flexibility. CNLs have successfully been used for writing requirements documents. We propose taking this approach well beyond this to fully functional programs. Through the use of heuristics and inference to analyze and determine the programmer's intent we are able to create fully functional mobile applications. The goal is for programs to be aligned with the way that the humans think rather than the way computers process information. Through the use of templates a **CABERNET** application can be processed to run on multiple run time environments. Because processing of a **CABERNET** program file results in native application program performance is maintained.

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

1 Introduction

Computer programming is about providing tools for improving the productivity of human users. The tools that are embodied in computer programs have effectively improved the productivity of all types of users. But one area that can still benefit from computer-based automation is the field of programming itself. 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 itself.

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" [2] that would handle many of the routine tasks required of programmers. These are but two of the extremes of researchers' visions for tools to aid program developers.

In Figure 1, we have captured the range of techniques that are involved in computer-assisted programming. These approaches include everything from requirements capture and code generation to evaluation of code for potential errors. 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 core function of the programmer, that of actual code generation.

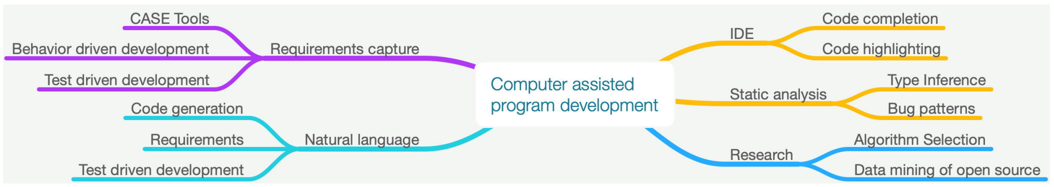

Fig. 1. Techniques for computer-assisted programming.

Much has been made of the use of Artificial Intelligence to replace human efforts. We see our efforts as not an attempt to replace the developer but as an opportunity to increase the developer's productivity. Our efforts follow in the path of Intelligence Augmentation proponents such as Doug Engelbart and Terry Winograd [3, 4]. To that end, the approach combines the use of 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 is still little progress on actual code generation. The challenge is to create a methodology that is flexible, intuitive, and thus, natural 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 proposed approach meets all of these requirements.

Our goals are two-fold. We seek to provide novice developers with a tool and approach which allows them to be productive without the significant barriers present 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 which we have named *CABERNET*. As part of our research we have developed a tool to generate mobile applications based on this methodology. In this paper we will describe the methodology and an example application built with it. In addition we will describe where we see this approach going in the future.

Through this research, we seek to determine if we can make significant improvements in programmer productivity and quality through a methodology utilizing inference and based on a concise and flexible controlled natural language-based programming tool.

2 Background

Researchers have long sought to automate various aspects of software development. Today there are many techniques available to help developers in their work. Some of these techniques are discussed below.

2.1 Requirements Capture

Requirements capture has seen significant 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* [5]. Requirements capture is an area where Controlled Natural Language approaches have previously been used [6]. For some years, the agile development community has sought to come up with better ways to capture user requirements. *Test-Driven Development (TDD)* was initially associated with agile development in Kent Beck's book on *eXtreme Programming* [7] and then expanded upon in his book on the subject [8]. 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* [9, 10] 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 [11, 12] and jBehave [13] are a couple of 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 be created in a traditional programming language.

2.2 Static Analysis

Static analysis tools come in a range of capabilities. The simplest of these tools are commonly referred to as lint tools [14]. These tools review the program code and identify violations of syntax rules provided for the target programming language. Violations can include punctuation, the spelling of reserved words, variables which 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 [15]. They have proved very useful in finding bugs on code which is already written. They help improve the quality code but do not help in the creation of the code.

2.3 Integrated Development Environments

The most commonly used tool for developers is the Integrated Development Environment (IDE). Tools such as Visual Studio, Eclipse, Netbeans, IntelleJ, PyCharm, Xcode, and others [16–21] provide a wide range of features to make the developer more productive. These include syntax highlighting [22], 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 [23], 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 particular syntax of the target programming language.

2.4 Declarative Syntax

Imperative programming [24] is the style utilized by most of the popular programming languages. These languages require the programer 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 each of those objects 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. Figure 2 contains the Swift code involved in creating a simple button that invokes a method called *processEach-PayThis*. 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 programmers goal.

```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)
```

Fig. 2. Swift code for Simple Button.

In 2019 Apple introduced SwiftUI [25] which utilizes a declarative syntax for describing the programs user interface. Declarative syntax [26] describes the results the programmer wants to achieve. Not how to achieve that result. Figure 3 includes the SwiftUI code required to create the same button as captured in Figure 2 but does it in seven lines of code, three of which contain only structural 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 you 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"))
    }
}
```

Fig. 3. SwiftUI code for Simple Button.

2.5 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 [27, 28]. NLC defined a natural language notation, which was interpreted directly to an output. In 1984 Knuth proposed *Literate Programming* [29], which combined $\TeX$ and Pascal to produce a vocabulary that had the primary goal of documenting for humans what the programmer desires the program to do. In 2000 Price, Rilofff, Zachary, and Harvey introduced Natural Java [30]. 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 to 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 [31]. Similarly, there have been efforts to define the user interface by extracting information from the natural language requirements documents [32]. The artifacts considered include error messages, variable names, and documentation in addition to the source code. 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 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 [33]. This approach represents another step along the path to natural language programming.

In another approach [34], 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 which humans leave unsaid. Much of human interactions are dependent upon shared experience and idioms, which allow humans to provide incomplete information and enables 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 [35]. 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 [36]. 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* [37], a Reasoning Engine [38] and a knowledge representation language as used to the Web Ontology Language (OWL) [39]. In these applications, Attempto has provided consistency to the tools by involving a specific subset of English.

Exman et al. [40] offer 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.

3 Our Approach

We have defined ***CABERNET*** **(Code generAtion BasEd on contRolled Natural languagE inpuT)**, an approach which allows a programmer to define a computer program using a Controlled Natural Language (CNL). Because the application domain is of limited scope and is specific to program definition, we can fill in the blanks using inference and implication. This approach provides a result similar to that experienced by typical human communications. We are seeking not to replace the developer in this process. Instead, our tool 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.

– Limited screen size

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

The use of constraint-based user interface design 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 the issues of screen size and proportions are handled through the use of templates, the programmer is freed from having to deal with those during development. Our approach allows the programmer to define an application for this popular platform with a simple human-friendly approach.

3.1 Basic Principles

The simplicity and directness of the approach are possible because there are many aspects of the design that 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 made use of a *broad set of defaults* that are applied when the developer omits the needed information from their descriptions. Second, we make use of *inference* to determine the developer's intent from the information provided (both within the user interface description and in the other artifacts that make up the program). Third, our approach allows for *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 from the default solutions that the developer accepts or declines, we build and reinforce our selection of recommended solutions. The characteristics of the proposed Controlled Natural Language model are as follows:

- Input language is forgiving
  - Outline based structure

  - Allow use of synonyms, acronyms and common abbreviations
  - Allow flexibility in ordering and location of descriptions
  - Minimum input required. In most cases, the input is keyword-based and does not require English sentences
  - Utilize popular Markdown [41] 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 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

4 Prototype

The use of a CNL means that multiple names can describe an object in the user interface. For example, in Figure 5 line 2, we refer to one screen of the application as a "Scene," and 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 accomidate this varied nomenclature we have implemented the concept of a thesaurous. The thesaurous captures a range of words that can be treated as synonyms. The contents of these thesaurous entrys come from multiple sources. First we have manually entered values based on our domain experience. Secondly we will populate the entries from web based thesaurous and searches of online documentation. Finally the thesaurous will continue to grow as *CABERNET* learns from programmers.

The approach utilizes the process shown in Figure 4. 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 of the objects. Then we utilize defaults and templates to fill in the missing parts of the description. In reviewing the program in Figure 5, you can see that, in most cases, the content of the bullets for each object 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 the majority of the outline-based program. In cases where a complex function must be executed (like the mathematical calculation in lines 45 through 47), the function is written in a more English-like format.

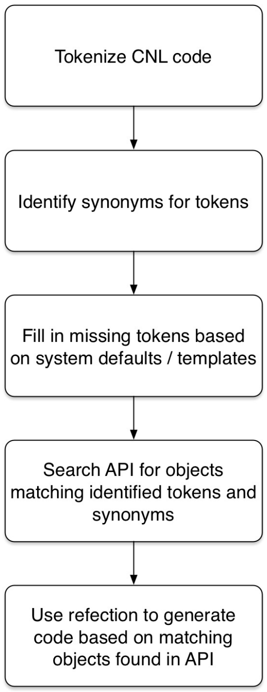
Tokenize CNL code
Identify synonyms for tokens
Fill in missing tokens based on system defaults / templates
Search API for objects matching identified tokens and synonyms
Use refection to generate code based on matching objects found in API

Fig. 4. Interpretation of CNL input.

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 also include items that cannot be identified as an object from the program. In this case, it is assumed that the item is not a function and rather text to be displayed. Examples of this include lines 19, 21, and 25. Given the issues with natural language processing, there can be more than one interpretation of the CNL form. In these cases, the developer is presented with a choice of options that the tool identifies as the most likely interpretation and possible alternate options. The results of this selection process is also an opportunity for the system to learn from these decisions. This feedback can be used for informing future choices and recommendations that the system produces.

4.1 Syntax

The notation for the Controlled Natural Language tool is based on the *Markdown* [42, 43] lightweight markup language. The input is structured as an outline, including only the information necessary to distinguish itself from the default. Some high-level rules that define the syntax include:

- #, ##, ###, 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

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 Figure 5. Line 1 of this code identifies the basic application. Lines 2 and 31 are one level indented from the application and start two distinct 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 format of the object. 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 category provide a default entry for the field.

```
# 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
```

Fig. 5. CNL code for example application.

4.2 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 format of the material. 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 Figure 5 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 the presence of mathematical operators 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 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 is 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**

Then, the remaining text is 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 evaluting the remaining text is to understand how to group the calculation. Mathematical expressions are usually evaluated from left to right adjusted by precidence rules and grouping defined by the use of 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"*. Clearly, 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 which values the multiply at the beginning of the line apply to.

```
( 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 Precentage* In evaluating these natural language expressions *CABERNET* converts these statements to traditional mathematical expressions with proper grouping of calculations. As a result *CABERNET* is able to 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 that *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 guideance. These would be an example of where the tranformation 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* is not able to 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 are not able to translate them into formulas.

“xxxxx-xxxx” or ***“(xxx) xxx-xxxx”*** or ***“mm/dd/yyy”***

Based on the extent of the evidence *CABERNET* will evaluate the likelyhood of the programmer’s intent being 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 which can be discribed in a *CABERNET* outline item. Using the same approach *CABERNET* is able to evaluate a wide range of program constructs. The steps in the process are as follows:

- Examine the natural language for evidence of type
  - words
  - symbols
  - contructs
- 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

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 in this fashion we are able to generate a working program.

5 Example Application

Figure 6 and 7 represents the output of our example application. In Figure 6, 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 tap-able 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 7 is the second screen of the application and includes a tip calculator. This screen contains 3 blanks to be 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, this final field is defined by the text in lines 45 through 47 in Figure 5. This calculation is triggered by tapping the “Calculate” button described in lines 34 and 35.

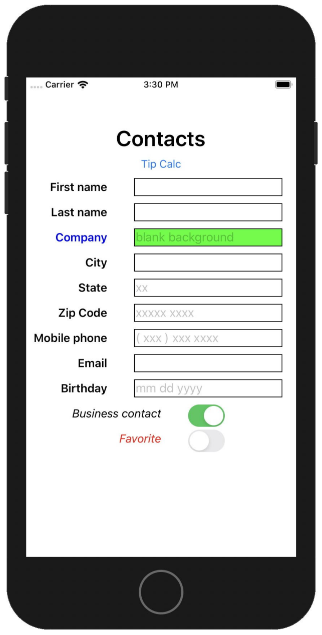

Fig. 6. App example, Addressbook.

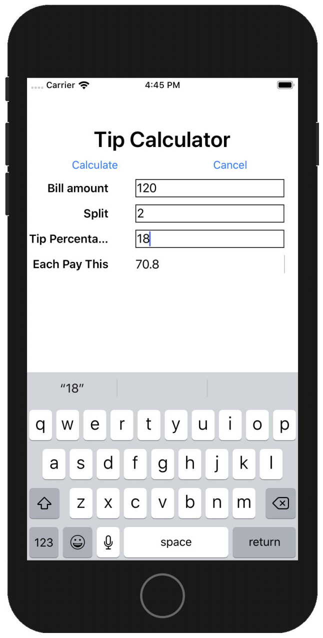

Fig. 7. App example, Tip Calculator.

5.1 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 proposed approach allows for expanding and customizing this information by *applying search techniques to the documentation* / APIs for the target development platform. By 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 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 the utilization of search techniques to develop a thesaurus for a new domain will help address this limitation.

6 Evaluations

The key process metrics that we seek to address with *CABERNET* include *code development speed, clarity, and size*. As can be seen, by the example we have outlined here, *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 shown in Figure 7 and the resulting Swift code to implement this program. This screen is implement by lines 31 through 47 of Figure 5, or 17 lines of *CABERNET* code. The resulting Swift view controller code that implements this screen includes 149 lines of code (almost nine times as much code). Now, much of this Swift code implements things that *CABERNET* handles as default values and constructs. But that is a large part of what this approach involves.

The other process metric that we seek to understand is our ability to expand the capability of *CABERNET* to address new target platforms and application domains. As we discussed above, we believe this is possible through the use of search techniques to create a thesaurus for these new targets. The information collected through this process should allow us to interpret the inputs in these new domains.

7 Future Plans

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

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 makes the development process much more approachable. By utilizing templates, synonyms, search of API, and reflection methodologies, this approach allows for CNL based code to generate applications for other platforms.

Future efforts will involve expanding the breath of functionality that is possible in applications developed with *CABERNET*. Our continued efforts will also evaluate the suitability of *CABERNET* for apppliation to new domains. As previously discussed, we plan on utilizing search techniques to harvest the required information from these new domains and application targets.

8 Conclusion

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. By using defaults and templates, the required code is *succinct and allows the system to fill in much of the details*. The result is a very terse source code with detail only included where needed to provide information for the program execution. As a demonstration, we have chosen a mobile device application design. With traditional methodologies, this is an area that can involve significant and complex code to document. Our approach significantly simplifies the required code *reducing the code size by a factor of 9 to 1* in one example. Not only is there significantly less code but the content of the program is significantly easier to read and understand.

This is a win / win proposition.

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

In summary, the ideal development tool for programmers of all capability levels.

References

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). 1970.
2. Terry Winograd. Breaking the complexity barrier, 1974.
3. Lawrence M Fisher. Siri, Who is Terry Winograd. `https://www.strategy-business.com/article/Siri-Who-Is-Terry-Winograd`, January 2017. Accessed: 2018-9-3.
4. John Markoff. *Machines of Loving Grace*. The Quest for Common Ground Between Humans and Robots. HarperCollins, August 2015.
5. K Beck, M Beedle, A Van Bennekum, and A Cockburn. The agile manifesto. 2001.
6. Agung Fatwanto. Specifying translatable software requirements using constrained natural language. In *2012 7th International Conference on Computer Science & Education (ICCSE 2012)*, pages 1047–1052. IEEE, 2012.
7. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.
8. Kent Beck. *Test-driven Development*. By Example. Addison-Wesley Professional, 2003.
9. Dan North. Introducing BDD, March 2006.
10. Behaviour-Driven.org. Behaviour Driven Software. `http://behaviour-driven.org`, 2016. Accessed: 2019-4-22.
11. Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.
12. cucumber ltd. Cucumber. `https://cucumber.io`, 2018. Accessed: 2019-4-22.
13. jbehave.org. What is jBehave? `https://jbehave.org`, 2017. Accessed: 2019-4-22.
14. P Louridas. Static code analysis. *IEEE SOFTWARE*, 2006.

15. Stefan Wagner, Florian Deissenboeck, Michael Aichner, Johann Wimmer, and Markus Schwalb. *An Evaluation of Two Bug Pattern Tools for Java*. IEEE, 2008.
16. Microsoft. Visual Studio. `https://visualstudio.microsoft.com`, 2019. Accessed: 2019-4-22.
17. Eclipse Foundation. The Platform for Open Innovation and Collaboration. `http://www.eclipse.org`, 2018. Accessed: 2019-4-22.
18. Apache NetBeans. Apache NetBeans: Fits the Pieces Together. `https://netbeans.org`, 2018. Accessed: 2019-4-22.
19. JetBrains SRO. IntelliJ IDEA: Capable and Ergonomic IDE for JDM. `https://www.jetbrains.com/idea/`, 2019. Accessed: 2019-4-22.
20. JetBrains SRO. pyCharm: The Python IDE for Professional Developers. `https://www.jetbrains.com/pycharm/`, 2019. Accessed: 2019-4-22.
21. Apple, Inc. Xcode 10. `https://developer.apple.com/xcode/`, 2019. Accessed: 2019-4-22.
22. Tanya René Beelders and Jean-Pierre du Plessis. The Influence of Syntax Highlighting on Scanning and Reading Behaviour for Source Code. *SAICSIT*, 2016.
23. Jian Li, Yue Wang, Irwin King, and Michael R Lyu. Code Completion with Neural Attention and Pointer Networks. *CoRR*, cs.CL, 2017.
24. Robert W Sebesta. *Concepts of Programming Languages*. Addison-Wesley, February 2015.
25. Jayant Varma. *SwiftUI for Absolute Beginners*. Program Controls and Views for iPhone, iPad, and Mac Apps. Apress, November 2019.
26. Chris Barker. *Learn SwiftUI*. An introductory guide to creating intuitive cross-platform user interfaces using Swift 5. Packt Publishing Ltd, April 2020.
27. Bruce W Ballard and Alan W Biermann. Programming in Natural Language: ”NLC” as a Prototype. In *Proceedings of the Annual Conference, ACM*, pages 228–237, New York, NY. ACM.
28. Alan W Biermann, Bruce W Ballard, and Anne H Sigmon. An Experimental Study of Natural Language Programming. *International Journal of Man-Machine Studies*, 18(1):71–87, 1983.
29. D E Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, January 1984.
30. D Price, E Rilofff, J Zachary, and B Harvey. NaturalJava: A Natural Language Interfaxce for Programming in Java. In *Proceedings of the 5th ...*, 2000.
31. Michael D Ernst. Natural Language is a Programming Language - Applying Natural Language Processing to Software Development. *SNAPL*, 2017.
32. Reyes Juárez-Ramírez, Carlos Huertas, and Sergio Inzunza. Automated Generation of User-Interface Prototypes Based on Controlled Natural Language Description. *COMPSAC Workshops*, 2014.
33. S P Overmyer, B Lavoie, O Rambow Proceedings of the 23rd, and 2001. Conceptual modeling through linguistic analysis using LIDA. *ICSE ’01 Proceedings of the 23rd International Conference on Software Engineering*, May 2001.
34. Mathias Landhaeusser and Ronny Hug. Text Understanding for Programming in Natural Language - Control Structuresz. *RAISE@ICSE*, pages 7–12, 2015.
35. Tobias Kuhn. A Survey and Classification of Controlled Natural Languages. *arXiv.org*, (1):121–170, July 2015.
36. Attempto Project. Attempto Project. urlhttp://attempto.ifi.uzh.ch/site/, 2013. Accessed: 2019-4-22.
37. Kaarel Kaljurand and Tobias Kuhn. A Multilingual Semantic Wiki Based on Attempto Controlled English and Grammatical Framework. *ESWC*, 7882(Chapter 29):427–441, 2013.
38. Norbert E Fuchs. Reasoning in Attempto Controlled English - Non-monotonicity. *CNL*, 9767(Chapter 2):13–24, 2016.
39. Rolf Schwitter, Kaarel Kaljurand, Anne Cregan, Catherine Dolbear, and Glen Hart. A comparison of three controlled natural languages for OWL 1.1. *OWLED 2008*, April 2008.
40. Iaakov Exman and Olesya Shapira. Fast and Reliable Software Translation of Programming Languages to Natural Language. *SKY*, pages 57–64, 2016.
41. S Leonard. The text/markdown Media Type. 2016.
42. S Leonard. Guidance on markdown: Design philosophies, stability strategies, and select registrations. 2016.
43. C Tomer. Lightweight Markup Languages, 2015.