LogicVis Project Proposal

CSE 403 Software Engineering
Winter 2019

Team LogV
Candice Miao
Leo Gao
Jed Chen
Glenn Zhang
Andrew Liu

Motivation

Do you remember a time that you have been programming for the whole day and ended up getting a headache from looking at the codes? Do you remember a time that you tried so hard to comprehend a piece of code with terrible style? Do you remember the time that you had trouble understanding code in a language that you are rusty with? The problem is that it takes a lot of effort for some programmers such as myself to truly understand a piece of code in our brains. Based on some scientific research -- [0](*Why we're more likely to remember content with images and video*), we process visual information 60,000-times faster than text. If there exists a tool that can help programmers visualize the code logic, it will be a lot easier for them to understand. Nowadays, there exist some text-based programming languages such as Java and Python, and there also exist graphical programming language such as Raptor; however, there does not exist a tool that can translate between these two types of programming languages.

Objective

The goal of our project is to build a tool that can help programmers visualize the code. To be more specific, we want a tool that can translate the text programming code we use today to a logic graph and also translate from a logic graph into programming code. If this tool is successfully built, programmers will be able to understand code with weird style or unfamiliar syntax much more easily. Furthermore, It does not only help programmers to construct the logic before they start coding, but also benefits debugging and testing process.

Current Approach

The most trivial approach to understanding a piece of code is to read the piece line by line. However, when the piece grows longer and longer, reading it line by line is extremely time-consuming and frustrating. People tried to solve this by abstracting from the actual code so that people will understand the code without actually looking at the code. There have been tools developed to generate documentation by executing the program. [1](Sulír, Matúš & Porubän, Jaroslav. 2017) Their approach comes in handy for people to understand people's code as a whole by looking at the documentation. But the logic behind the code is still not understandable unless the actual code has been read, and this will be difficult for people who need to modify the code for other purposes like adding features and debugging. There are also graphical code visualizers like CodeSonar that visualizes the relationships between function calls [2], and SourceTrail that

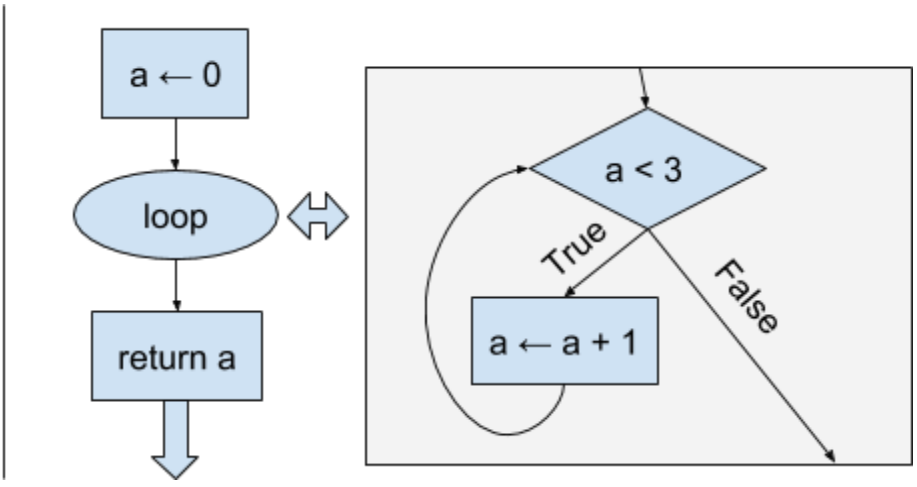
visualizes the structure of the class [3]. However, none of these emphasize the logic within the code that is essential for understanding and modifying the code. Users still need to look into the code to be able to understand the logic behind the code.

Our Approach

LogicVis is a program that takes an input of a piece of Java code and outputs a logic graph. It reads in the Java code line by line and turns each line into a node representation with the shape indicating the type of command. LogicVis graphs contain different representations for basic Computer Science concepts, including if/else, loops, etc. We will first implement a method that takes in a piece of Java code containing a single java function and outputs a built graph object that contains the logic visualization for the function as our Minimum Viable Product. Then we will work on implementing the Graphic User Interface so that it is easier for Computer Science beginners to utilize. During the last few weeks, we will work on expanding our input range to taking a class rather than a single function so that users can also visualize the relationship among functions within the class. Our final product design is to have a functioning Graphic User Interface that takes in a piece of Java code (one or multiple functions, or a class) and outputs a logic graph that is constant and organized with reasonable usability and readability. If we have time in the end, we want to potentially look at implement this product as a plugin to Eclipse or supporting modification on the graph generated being reflected in the code. Since we are not planning on basing our product off of any existing products or purchasing outside resource, our cost plan, in this case, is 0. We have allocated 8 weeks based on our designed functionality of the final product. However, we did also account for estimation errors by designing a Minimum Viable Product that will be implemented first and a stretch goal in case we have time.

Example:

```
int a = 0;
while (a < 3) {
    a++;
}
return a;
```



Limits/Risks

When designing this software, we will face a decision that is crucial to the success of our product. That is, we need to balance the clarity of our design with the amount of detail we put into our graphs. On one hand, we want to represent each line in a function in the most understandable and informative way. However, combining this with the length of functions, which can go over a hundred lines, and the fact that a class could contain multiple functions, it is easy for our graphical interface to get cluttered to the point that our software is ineffective. On the other hand, if our graph is too simple, it may not be any easier to understand than the code itself. In order to find the perfect balance, we will do research to find out which types of statements and declarations users find most important and which ones they find most confusing. This way, we can build the most optimal graphical interface for the user.

We have also taken risks in deciding the scope of our project. We have chosen to make this software specifically for Java, which means that building a graph for any other language is not possible. Additionally, this means that we should be careful to properly parse Java keywords with our software. If any bug occurs during parsing, the output graph would absolutely fail to convey the proper meaning to the user. We intend to perform extensive testing after each checkpoint in our software, as detailed in the schedule in the next section. Another risk that we will be taking is our decision to not handle Java built-in library functions. Displaying these would introduce a whole new set of problems, such as additional clutter and having to implement it in the first place. So, we currently have chosen to treat imported functions like a generic line, but we may change this depending on the feedback we receive when doing research. Overall, we hope to build our software for a large enough scope of statements to help the user while not also overwhelming them. If we have successfully chosen the correct scope for our product, we will have made the version of our software that will help most people.

Schedule

We plan to split major sections of the projects into work done for each week. For each week, we will begin planning the specifics of what each goal entails. After reaching the goal of implementation for every week, we start testing the programs to ensure its correctness. This means that testing will be involved in every week, so we chose to omit it in the schedule below.

Week 4 goal:

- Project proposal and Planning

- Write and finalize the initial specification
- Design visual components of graphs

Week 5 goal:

- Function parsing: Identify the different type of lines in Java

Week 6 goal:

- Build a data structure to represent the code

Week 7 goal:

- Visualize the data structure into a readable graph
- Design the GUI

Week 8 goal:

- Research on popular GUI components by experimentation
- Build the GUI

Week 9 goal:

- Handle multiple functions
- Link relations between existing functions

Week 10 goal:

- Parse class structures
- Handle multiple classes

Week 11 goal:

- Final testing
- Prepare for presentation

Stretch goal:

- Make it a plugin for Eclipse
- Support modification on generated graphs being reflected in the code

The minimum viable product with the GUI will be done at week 8. The final product, which includes processing multiple classes, will be done at week 11. A good “midterm exam” for this project would be checking to ensure that the base visualization of the graph we are building matches our expectations at week 7. A good “final exam” for this project would be looking at the final product to see whether we are handling classes correctly at the end of week 11. At that point, we would also test everything else to see that they are robust and optimal rather than just its correctness according to the specification.

Testing

As described above, we would be testing constantly as we start implementing each part of the project. These tests mostly check that the program behavior matches what we expected. They include verifying outputs, memory usage, time consumed, and how well

it contributes to our overall purposes. Though the first three parts can be concretely measured with our written tests, the last part requires extensive experimentation on other people's opinions on our program. Our basic purpose to help programmers understand the code better through the use of graph visualization. To validate that our project achieves its goal, we will perform the following experiments:

1. Code vs. Graph: Put a regular 20 line-long program in code form and graph form. Then ask customers to compute the output of such programs giving them only 1 form versus giving them both forms. We would measure the correctness of the answers and the time taken by either side to check whether the graphs actually helps with understanding. The following is a sample graph. The points scored will be a system based on a linear combination of correctness and timing, which will be determined after we find out what the performances look like.

2. Graph vs. Graph: Put a variable length program in code form and different graph forms. Then ask customers to compute the output of such programs giving them the code and one of the graphs. We would measure how the customers performed using correctness and time spent to check which interface of our program is more intuitive and effective to use. The following is the other graph, where we will use a point system similar to the one used above.

Using these kinds of experiments, we can evaluate our approach and determine whether changes are needed.

Citations:

[0] Rachel Gillett (2014). Why We're More Likely To Remember Content With Images and Videos (Infographic)

[1] Sulír, Matúš & Porubän, Jaroslav. (2017). Source Code Documentation Generation Using Program Execution. Information. 8. 148. 10.3390/info8040148.

[2] https://www.grammatech.com/products/code-visualization

[3] https://www.sourcetrail.com/