Exploring the Use of Macros for Generating Code Suggestions in the Context of Code Bubbles

Contents

INTRODUCTION

"Code Bubbles is a front end to Eclipse designed to simplify programming by making it easy for the programmer to define and use working sets. Working sets here consist of the group of functions, documentation, notes, and other information that a programmer needs to accomplish a particular programming task such as adding a feature or fixing a bug." This is the official description for Code Bubbles, as inscribed in the official Brown University website: http://cs.brown.edu/~spr/codebubbles/

Code Bubbles is an advanced tool which aims to aid developers in their programming tasks. This Capstone is centered on one aspect of Code Bubbles that was developed in the spring semester of 2019 by undergraduate student Dimitrios Parthenopoulos, under the guidance of professor and developer of Code Bubbles, Steven Reiss. The project aimed to incorporate macro technology in order to offer code suggestions to the user of the software. Macro in computer science is defined as a pattern that determines how an input sequence should be mapped to a replacement output sequence according to a defined procedure. The macro in this case was the following: When using Code Bubbles and creating programs using the bubbles of the software, a macro is required to predict what lines of code will follow an input line of code, given the code that has already been written in previous bubbles. A bubble in the context of Code Bubbles is a front end interface that can be used as an IDE. Users of the tool use bubbles to create classes, methods and interfaces, as well as populate the bubbles with code that can be compiled and run.

This paper examines the techniques used in order to create this macro as well as the procedure that took place during these five months of research.

PHASE 1

The first phase of the paper lasted from January 2019 until late February of 2019. During that time the main focus was to install the environment of Code Bubbles in a local computer as well as make the code compatible to run with Java 8. That required lots of hours of work spent in updating source files and resetting scripts so that they would allow the environment to run with the updated Java version. Another aspect of that process was the adjustment to a local device that was not being hosted by professor Steven Reiss. A lot of the files required special access so work had to be done in order to establish an accessible environment at which a developer could make changes to the files of the project and see their changes reflected on the project itself. This process took the better part of a month but it was essential since it meant that future developers would have the ability to work on the project without the need to reestablish all of these settings.

During the first phase of the project, parallel to working on the installation of the environment, this was the time that research around the topic of macros was conducted. The 2 papers studied were Miryung Kim's paper: "Discovering and Representing Systematic Code Changes"[1] and their other publication under the name "Automatic Inference of Structural Changes for Matching across Program Versions"[2] Those two papers dealt a lot with a technique regarding optimized post processing of matches found in a macro. That technique explained an algorithm that can be used in order to sort and evaluate the matches that have been found for a

[1] http://web.cs.ucla.edu/~miryung/Publications/icse09-lsdiff.pdf
[2] https://homes.cs.washington.edu/~djg/papers/matching_icse07.pdf

specific input and the way that they can be presented in terms of their importance. This technique includes the use of the mathematical equation: $S=M \cup L/Tm$, where S is the significance of a result, M is the number of matches it has, L is the set of results that are being considered and Tm is the total number of matches for a specific input. Essentially that formula defines the significance of a result as a positive relation to how many matches it has and in a negative relation to the total number of sentences that are being examined. This formula provided inspiration to the technique used to sort the matches for suggestion in the Code Bubbles model.

PHASE 2

The second phase of the project involved the implementation of the macro technique in order to provide suggestions. Code Bubbles has a feature that allows users when right clicking on a logic bubble with code, to retrieve suggestions for what lines of code should follow the code that is being highlighted by a cursor. The development of that feature was the aim of the second phase.

In order to develop suggestions based on all code written the first step was to take every single logic bubble in a user's project and decompose all of the user's code that is written inside it. This is done with the use of the Java structure known as ASTNodes. Inside the BpareTrie file, logic bubbles are broken into ASTNodes. This is done in the following way: The logic bubbles are broken down in compartments based on the logic phrases that they contain. Meaning that for a logic bubble, there is an ASTNode containing all of the code in the bubble. The next ASTNode breaks the bubble into classes and each class is made into an ASTNode itself. Then, BpareTrie identifies any braces such as { } which signify that there is a statement like a for loop, a while loop or if statement. Once { } are met there is another ASTNode made containing only the code

inside those braces. This process is repeated until no further break down into ASTNodes can be made. To provide an example consider the following logic bubble:

```
Class Car implements Vehicle
{
        Public Car(int type){
                if (type==1){
                        System.out.println("This is a Ford");
                }
                else if(type==2){
                        System.out.println("This is a BMW");
                }
        }
}
```

This would be broken down in the following ASTNodes:

Node 1:

```
Public Car(int type){
        if (type==1){
                System.out.println("This is a Ford");
        }
        else if(type==2){
                System.out.println("This is a BMW");
        }
}
```

Node 2:

```
if (type==1){
        System.out.println("This is a Ford");
}
```

```
else if(type==2){
        System.out.println("This is a BMW");
}
```

Node 3: System.out.println("This is a Ford");

Node 4: System.out.println("This is a BMW");

After establishing an understanding for all ASTNodes and the process at which they are formed, a way had to be found to organize all of them. In the private static class MatchResult, a private instance of Map was created that mapped ASTNodes to double[]. This map is iterated through in the printResult method, and every single ASTNode is presented. The value_map is being returned by a getMap() method so that it can be accessed in a different class. The PriorityQueue<String> match(ASTNode n) method in BpareTrie is where the macro is being created. This method takes in an ASTNode, namely the logic block that the user wants to generate suggestions for. When right clicking in a piece of code in Code Bubbles and selecting "get Suggestions" that piece of code is being input in the match method as "n" and the process begins to find its matches and generate suggestions for it. During this process this ASTNode is compared with every single other ASTNode that has been generated by all logic bubbles, against various parameters, using string comparison and other techniques. If a match is found, this implies that there is another part of the code that contains the same commands as the ones the user is trying to find suggestions for. Upon finding a match, the program looks at the logic bubble of the match and determines which ASTNodes in that bubble could be viable suggestions based on their context with the match itself.

Once suitable suggestions are found they are added into a PriorityQueue<String>, whilst keeping track in an integer, of how many times they have appeared so far in this process. The

PriorityQueue stores the suggestions in a String form. The choice to store suggestions as Strings instead of ASTNodes was done because Strings contain built in methods that make it much easier to compare them with one another and break them down into their characters, than ASTNodes. The integer associated with the string determines how many times these suggestions have been encountered and thus determines which suggestion should appear first in the PriorityQueue. The PriorityQueue presents suggestions in a descending order, from the most relevant to the least. The process of finding suggestions ends when all available ASTNodes have been examined and have been compared with the original input "n" that the suggestions are for. Once the final PriorityQueue has been formatted and the strings are stored based on their frequencies, the PriorityQueue is sent to a different file, BpareMonitor.java. In this file, in the string getSuggestion method, a BpareTrie is created from the BpareProject. That BpareTrie contains the PriorityQueue which has all of the suggestions. That PriorityQueue is retrieved using the match method of the BpareTrie. If there are no suggestions to be made, a JOptionPane is created in the Code Bubbles front end, displaying the message that "There are no available suggestions at the time". If the PriorityQueue contains elements, using an iterator these elements are processed one by one, where they are treated as individual strings, eventually all of them being added to one final message string, which is displayed through a JOptionPane to the user when they click the "Get Suggestions" button. This completes the entire task of the project, as it successfully manages to retrieve suggestions to the user, sorted based on their frequency of appearance.

In terms of performance the model operates at good speeds, allowing the user to get the results they want without needing to wait for prolonged periods of time. The worst case runtime of the match function, which contains the bulk of the logic that goes into finding the suggestions

is O(k* klogk). k represents the number of total ASTNodes that are generated. Since all of the nodes generated are examined through a loop, the initial runtime is O(k). Since in each iteration the PriorityQueue is sorted in order to contain the strings in a descending order of frequency, that process takes O(k*logk), so overall the runtime is O(k*klogk).

CONCLUSION AND FUTURE IMPROVEMENTS

Overall this project can be described as successful. The result was met and the outcome was positive. The goal to allow the user of Code Bubbles to retrieve suggestions based on their preexisting code was achieved. The question now becomes how useful this will be for users of Code Bubbles and how much this feature will be utilized.

In terms of future improvements one aspect of this capstone that could be improved is the following: The user is presented the suggestions in a JOptionPane. This could be done more efficiently if the user had the option to click on the suggestion when it appears on the screen and the suggestion would be automatically be incorporated to the user's code, at the point where the user selects. This would improve the automation of the whole process significantly and would be especially useful if the user was dealing with suggestions which would take tens or even hundreds of lines of code. The reason why this feature was not implemented even though it was identified was purely due to time. The project spanned a single semester and during that time the focus from an academic perspective was chosen to be on back end development of macros, instead of front end features which could be implemented at a later date.