OkEclipse: Voice command based Software Development

Vismay Golwala
NC State University
vjgolwal@ncsu.edu

Dharmang Bhavsar
NC State University
dmbhavsa@ncsu.edu

Srija Ganguly
NC State University
sgangul2@ncsu.edu

Tushita Roychaudhury
NC State University
troycha@ncsu.edu

ABSTRACT

Natural Language Processing has impacted some of the major industries [6] resulting in some superlatively innovative changes. But the software development process has still been largely untouched by this powerful technique. Taking this into consideration, we try to incorporate some Natural Language Processing features into the already existing feature rich OkEclipse plug-in. We try to extend the use of NLP from just development phase in the Software Development Life Cycle to the Testing phase as well. New features helping programmers work with multiple languages are also planned to be added in our implementation of the project.

Keywords

Eclipse, Java, Python, voice command, testing, version control, code generation

1. INTRODUCTION

"Voice based softwares" are a thing right now. But what about "Voice based software development"? We ask. OkEclipse is an answer to that. The system worked in helping programmers code by giving them features like code searching, error searching etc directly in the development environment. Although the system is working effectively, we saw a lot of opportunities to make it better. So, our efforts aim to expand this work by integrating voice based commands throughout the software development life cycle. Also, some more work is put in to make programmer's life easier by making features such as code conversion, code generation and voice based version control. Previous efforts in integrating NLP in SDLC deemed to be successful.[1][2][3] The main aim is to start a "voice based innovation streak" in the all the phases of SDLC in the same way that that voice recognition technology has affected other domains.

2. EXISTING SYSTEM

In the first phase of this project, OkEclipse intended to enhance the user experience of the Eclipse Integrated Development environment by incorporating speech recognition. The current OkEclipse implementation has the following features:

- Listening Menus: Menu options, even when highly nested, can be invoked with the user reciting the option label text. Upon receiving voice input, the text-equivalent is matched with a lookup table to retrieve the corresponding Command ID. Then this request is passed to the eclipse command framework to invoke the said command.
- Loud Console: Whenever the user needs a recommendation regarding the issues listed in the console log, the user can invoke the "Find" command to get suggested solutions from crowd-sourced Q&A repositories and also relevant YouTube videos to troubleshoot the issue.
- Sound Programmer: This feature aims to reduce the time spent on certain frequent and mundane tasks like generating getters and setters, creating the main method, writing the toString function. These tasks have been implemented to be achieved by voice commands.

3. USER SURVEY

This section describes the questions that were asked in the user surveys, the reason for asking those questions, the results and the conclusions that we derived from the results.

Question 1: **How do you feel about being able to manage your github repository directly from the Eclipse IDE using voice commands?**

The reason for asking that question was to get a feel of whether integrating GitHub version control support in the Eclipse plug-in. From the first iteration of the project, we personally found that it was inconvenient to open command line and then push the code to GitHub even if the project was already cloned in the existing directory.

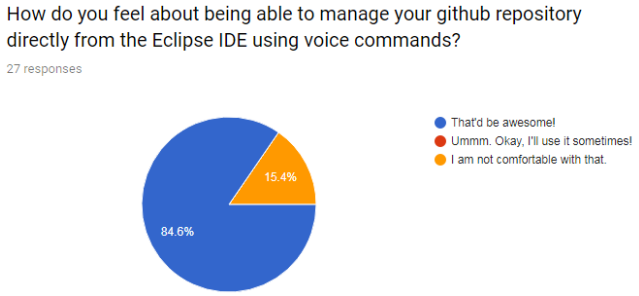

Figure 1: Responses for survey question 1

The results in Figure 1 are interesting as a majority of users ~85% wanted such a system. They might be going through the same inconvenience that we did go through in the project.

Question 2: Would it help if Eclipse IDE generated code syntax of different constructs, depending on a chosen language, via a simple voice command?

The question was aimed at knowing whether people were interested in a system that generated code according to certain requirements. Code generation is no mean feat (if it was, why we would be studying CS).

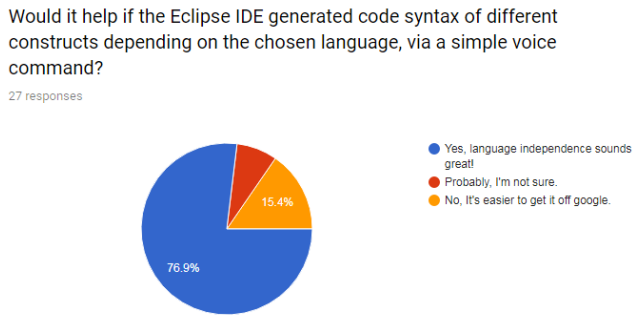

Figure 2: Responses for survey question 2

Again the results were largely positive (Figure 2), with some unsure and some negative reviews. This indicated that people would like such a feature to be present in the IDE itself rather than going to Google to search about that.

Question 3: How would you like voice commands that automate unit testing of your code?

As we said, we were trying to integrate voice command features into different SDLC life cycle phases. This was an attempt to get to know the sentiment of public for such a system. From Figure 4, it is evident that the response was totally positive and was the highest ranked feature that surveyors wanted.

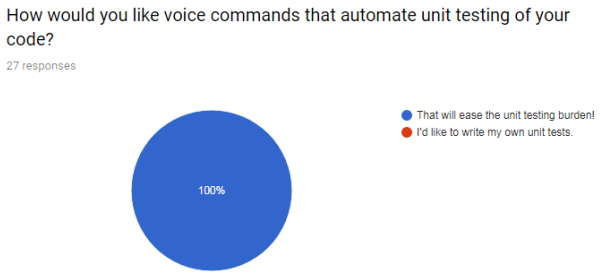

Figure 3: Responses for survey question 3

Question 4: Out of all the features mentioned above, please rate them from 1 - Most Preferred to 5 - Least Preferred

We were trying to garner the popularity of each feature in order to make the popular feature easiest to use and highly robust so that it has wide usability.

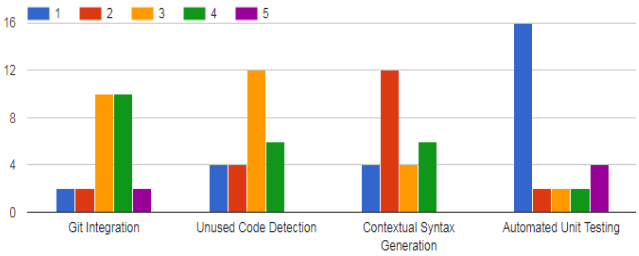

Figure 4: Responses for survey question 4

4. IMPLEMENTATION

4.1 Use Cases

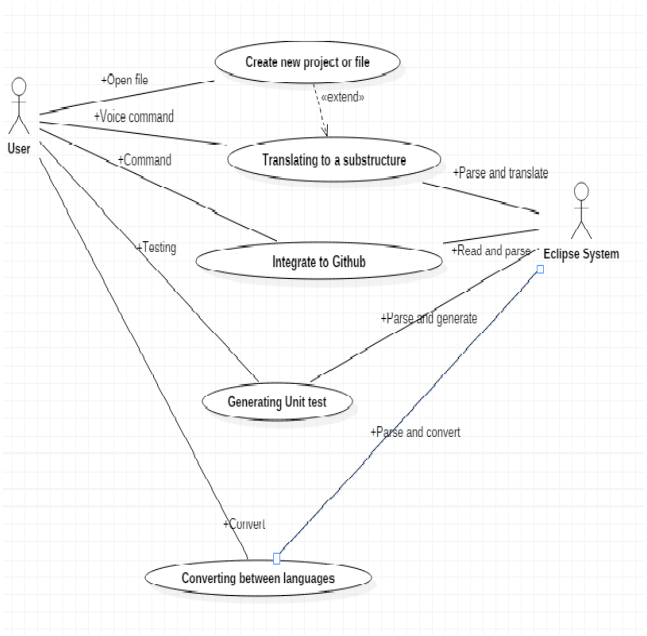

Figure 5: Use case diagram

4.1.1 Use case 1

Use Case Name	Translating to a substructure
Actors	User , Eclipse System
Description	User will voice command to generate a substructure. Eclipse system will parse the input and generate the structure.
Trigger	Use case is triggered when the user selects voice command feature and speaks out a command having similar syntax
Preconditions	1. User has a PC 2. User has active internet connection 3. User has Eclipse 4. Project has been created
Post conditions	1. Substructure syntax created
Normal Flow	1. User creates new project 2. User selects voice command feature and speaks 3. System interprets
Special Requirements	Sneeze and other noise needs to be controlled
Assumptions	User understands and speaks English
Notes and Issues	Interface is basic and provides a trigger to the main backend work of the software

Table 1: Use case 1

4.1.2 Use case 2

Use Case Name	Maintaining version control
Actors	User , Eclipse System, Git, GitHub
Description	The user would want to get the latest code or push the changed code to central repository
Trigger	Use case is triggered when the user provides a command asking for git commit, pull or push
Preconditions	1. User has a PC 2. User has active internet connection 3. User has Eclipse 4. User has Git
Post conditions	Eclipse pushes the changes to GitHub or fetches the codebase from there
Normal Flow	1. User creates a project and starts coding 2. User invokes commit, push or pull command
Special Requirements	Noise and sneeze needs to be kept in control
Assumptions	User understands and speaks English
Notes and Issues	Interface is basic and provides a trigger to the main backend work of the software

Table 2: Use case 2

4.1.3 Use case 3

Use Case Name	Generating unit test
Actors	User , Eclipse System
Description	Unit testing cases are generated automatically
Trigger	Use case is triggered when the user speaks a command that asks for generating unit tests
Preconditions	1. User has a PC 2. User has active internet connection 3. User has Eclipse
Post conditions	Eclipse shows an updated project with test cases
Normal Flow	1. User finishes developing the application and wants to test it 2. User selects voice command and speaks for unit testing 3. System generates unit test
Special Requirements	Voice commands require some specific sensitivity
Assumptions	User understands and speaks English
Notes and Issues	Interface is basic and provides a trigger to the main backend work of the software

Table 3: Use case 3

4.1.4 Use case 4

Use Case Name	Converting between languages
Actors	User , Eclipse System
Description	The user will be able to convert between languages like Java to Python for basic structure
Trigger	Use case is triggered when the user selects or chooses the command to convert to another language
Preconditions	1. User has a PC 2. User has active internet connection 3. User has Eclipse
Post conditions	Eclipse shows a new project with the substructure reflected in a different language
Normal Flow	1. User commands code conversion 2. User provides a new file name 3. System generates current code structure in the required language
Special Requirements	Voice commands can be susceptible to noise and code generation should be of simple code written
Use Case Name	Converting between languages
Actors	User , Eclipse System
Assumptions	User understands and speaks English
Notes and Issues	Interface is basic and provides a trigger to the main backend work of the software

Table 4: Use case 4

4.2 Enhancement and New Features

Specifically four more features were added to the prior version of the OkEclipse plug-in in this iteration of the project. The prior version focused on using voice commands for the coding of the software development life cycle. In this iteration, we try to take that up a notch by including things like code conversion, unit testing, code generation and git integration. The details are as follows:-

1. **Code Generation** - Code generation generates code according to the specifications mentioned by the user. The voice command for code generation is to say “generate” and

then continue with the specific voice command that states the type of code one needs to generate, which is something like “generate - a function abc with three integer variables”. This voice command will generate a function named “ abc “ and will have three parameters of int type passed to it. Another way to generate code in the plug-in is to write the specification of the code that needs to be generated in between two “$ $” (dollar) signs and pressing “Shift+G”. The definition of the code that is generated can be changed by the user according to his/her will. Our code generation generates basically three types of code constructs - classes, functions and loops. The commands for all of them are different according to the type of construct to be generated.

- class generation - “generate - class CARS with 3 private variables”. Output of the command is shown in Figure 6.

```
class CARS{
    private String  a, b, c;
    CARS(){
    }
}
```

Figure 6: Class generation output

- function generation - “generate - function FindMax that has 3 int parameters and returns int”. Output of the command is shown in Figure 7.

```
public static class class_FindMax{
    public static int FindMax( int a, int b, int c){
        return 0;
    }
}
```

Figure 7: Function generation output

- for loop generation - “generate loop for 23 iterations”. Output of the command is shown in Figure 8.

```
for(int i=0;i<23;i++){
    System.out.println(i);
}
```

Figure 8: Loop generation output

The basic things such as integer and string can be replaced by any other primitive data type and the number of variables can be changed to any amount you want up to 100.

The code generator works as follows. Firstly, the voice commands are caught by the Sphinx system and are converted into a string. Then we parse that string to know what kind of generation the user wants. A simple Java program to generate the required syntax is then called with the parameters that are parsed from the input and it appends the code to the end of the program. The code that is appended is indented by our own module so no care needs to be taken regarding indentation. The variables that are generated inside a class or a function call are named a,b,c and so on.

2. **Code Conversion** - The motivation behind implementing this feature is to give the user a kickstart if there is a requirement for conversion or migration from Java to Python or the user wants to learn Python and already knows Java. Code conversion converts the code from Java to Python with just the keywords “convert”. No other actions are necessary. The code conversion will make a new file named source.py in the same directory where the java program was stored. Output is shown in Figure 9. Depending on the type of Eclipse installed, it may or may not let the user .py files so this file needs to be opened in other code editors if possible. If the converted code needs some Python libraries to be called, the libraries imports them in most of the cases if the libraries are in the basic package of Python itself. Though no guarantee is taken for the converted file to be running or correctly running, the extension we used provided with some warnings as to where there might be some problems in the converted programs.

Main.java

```
public class Main {
    public static void main(String[] args) {
        Main m = new Main();
        System.out.println(m.divide(10, 5));
    }

    public float divide(int a, int b) {
        return (float) a / b;
    }
}
/*
#!/usr/bin/env python

""" generated source for module source """
class Main(object):

    """ generated source for class Main """
    @classmethod
    def main(cls, args):
        """ generated source for method main """
        m = Main()
        print m.divide(10, 5)

    def divide(self, a, b):

        """ generated source for method divide """
        return float(a) / b
*/
```

Figure 9: Code conversion output

Code conversion uses a Python library called java2python which converts Java code to Python2 code. The library has an extension which will show a warning when the code is generated with the exact line which it thinks will not run. These warnings are shown in the command line when the java2python library is called. The code conversion that happens might not be 100% correct, that is the reason it shows warnings. The warnings shows the line at which the code might not run and the reason for that. The java2python library needs Python2 to run and needs ANTLR 3.1.3 runtime of Python. Since we did not want users to install more things on their local machines to run code conversion, we implemented the conversion on a server. The API calls for the server was also made from scratch by us. The full file that is to be converted is sent to the server and the code conversion happens there and then the file is sent to the local station back again and saved in the same working directory. The API calls were implemented in Python using the Flask API and some of the standard libraries in Python were used (like sub process) to call the command line and trigger the code generation using the java2python library. The warnings when the code conversion was not 100% runnable were displayed in the console on server, so they

had to be fetched as well along with the converted code. These warnings are shown in the Eclipse console.

3. **Unit Testing** - Unit testing is the flagship feature of our application because it was the most asked feature in the evaluations. Unit testing in our plug-in uses no extra library or API, the whole functionality is implemented from scratch by the team itself. To unit test a particular function, the full definition of the function should be selected and then the voice command "testing" or shortcut of "Shift +T" does the magic.

```
public void unitTesting() {
    try {
        float x;
        x = divide(10, 2147483647);
        x = divide(-2147483648, 0);
        x = divide(1, 1);
        x = divide(10, 0);
        x = divide(-10, 5);
    } catch (Throwable e) {
        e.printStackTrace();
    }

}
```

Figure 10: Unit testing output

The testing functionality first parses the definition of the function and makes a list of the parameters that are present in the function. Now after finding that, the unit testing module makes a function in the program itself and tests the functions on different values of the parameters. It specifically chooses the boundary values of the parameters and tests the function on other 100 random values as well. This 100 random calls can be changed to make any amount of calls by changing the value in the 'for' loop of the function. Output of testing is shown in Figure 10. After the unit testing function is created, the user needs to re-run the program and the results of all the calls on the function will be displayed in the console of Eclipse. The system will catch errors and exceptions as well during the process and show them in the Eclipse console. The function works with all basic primitive data types as well as strings and arrays, but not on user defined data types. If the function you choose uses a user defined data type, then out unit testing library will not work. You could expect super fast performance from our implementation as all the code is native, runs on local machine and is just a re-run of the program with different calls on the subroutine with different parameters.

4. **Git integration** - Another functionality of the plug-in is version control integration, specifically GitHub integration. The various functionalities of GitHub that our application supports are status, commit, push and pull. The four functionalities are developed separately and they have their own voice commands and shortcuts.
    - git status - Say "status" or press "Shift + S". Output is shown in Figure 11.

```
Added: []
Changed: []
Conflicting: []
ConflictingStageState: {}
IgnoredNotInIndex: [edu.ncstate.csc510.okeclipse/bin, Git Integration/bin/JGitPrintContent.class, Voice Integration/bi
Integration/bin/CloneRepository.class, Git Integration/bin/CommitRepository.class, Git Integration/bin/ShowStatus.clas
Missing: []
Modified: [edu.ncstate.csc510.okeclipse/src/edu/ncstate/csc510/okeclipse/handlers/SoundProgrammerImpl.java]
Removed: []
Untracked: []
UntrackedFolders: []
```

Figure 11: Git status output

    - git commit - Say "commit" or press "Shift + C"
    - git push - Say "push" or press "Shift + P"
    - git pull - Say "pull" or "Shift + U"

The implementation makes use of the JGit library in Java for Git integration. The library needs some parameters to run. These parameters are -link to the git repository, -user email ID and -password. The input from the user for the above parameters are taken by JGit and sent to Github. Then git searches for the".git" folder in the current working directory and performs the commands accordingly. It can performs all the functions mentioned above seemlessly with just a prerequisite, there should be an already initialised git in the current working directory to work on, which creates the ".git" folder in the current working directory. The final results when the code is pushed, pulled or commited is shown in the eclipse console.

4.3 Software Engineering

We followed a certain mix of the Agile methodology and test driven development while developing this project since this project was a time sensitive situation which we had to complete in about a month. This helped us in building and integrating the plug-in incrementally and without any big hurdles along the way.

We held discussion sessions as well for brainstorming if anyone faces some serious problems along the way. During one of such meetings we came up with a plan to use our own testing module and to use a server to do code conversion. Extensive study on the proposed features and estimating the time that could be required for implementing these features helped in the long run as we were able to capitalize on the time available and did not run into some serious time constraints. We also performed code reviews during integration so that everyone understands what was done and how to exactly integrate the thing. The test driven development methodology helped us in understanding where we stand and which features work, to what extent do they work and for what cases they do not work.

We primarily divided the work between the members according to the features and performed sprints in which we implemented each of our modules and delivered on the final result smoothly. We called this the divide and conquer strategy. The features divided among the members are shown in Table 5:

Features	Names
Code Generation	Srija, Dharmang
Code Conversion	Dharmang, Vismay
Git integration	Tushita, Srija
Unit Testing	Vismay, Tushita

Table 5: Division of work

4.4 Technologies used

Our system uses simple and easily available technologies. The description of the languages, environments, API and libraries used are as follows. The application is an Eclipse plug-in so it is majorly built in Java. There are some components in the application that use some other languages such as Python.

1. **Java** [16]**:** It is used for the core plug-in development, accessing the file system, making API calls and performing programming for inputs to process and derive results.
2. **Python** [15]**:** It is used for developing an API using flask to convert the given Java code into Python.

3. **Sphinx** [7]**:** Sphinx is the Open Source Library which provides keyword based speech recognition. We have integrated Sphinx into our system to handle triggers for different keywords by user.
4. **JGit** [10]**:** It is the library used for enabling the Git and GitHub integration into the system. The library provides inbuilt methods to call the commit push and pull functionality. It accesses the user's remote URL and login credentials for pushing the code base.
5. **Java2python** [9]**:** This library is used to convert user's code from Java to Python. It is used in the project hosted on the Digital Ocean Server [13].
6. **Eclipse** [4]**:** Eclipse is the environment used for plug-in development and also the execution of the system.

5. ARCHITECTURE

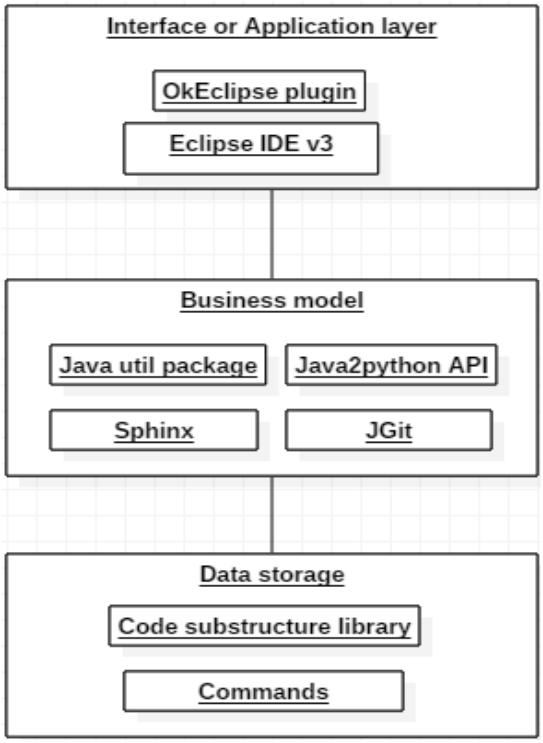

Figure 11: System Architecture

The architecture of the system (Figure 11) describes the required components, plug-ins and APIs used to make the current version run and it states a three tiered architecture for the application.

The Interface layer uses the properties of the Eclipse environment and the OkEclipse existing plug-in to display outputs. The plug-in contains all the required libraries for the system. During the startup of Eclipse, all these libraries are loaded into the environment of user and can be accessed by the compiled class files of the plug-in. The plug-in is developed for the **Eclipse IDE v3** and the older versions might have conflicts in the features provided.

The functional layer uses four important components to achieve the features described. **Java util package** is used in all the existing features especially code generation. It is used for string manipulation, data structures access and iteration over the existing code skeleton to form the output Java code. **java2python** is a library that is used to convert the code from Java to Python 2. It is hosted on the Digital Ocean server where there system is set up using Python and its Micro Framework Flask. **Sphinx** library is used for speech recognition and the command library built calls the existing Java handler classed during the triggering of functionality. **JGit** library is used to implement rudimentary Git functionalities and integrating them into the system.

The data store layer is used to refer to certain stored operations like **code substructure** store contains the basic skeleton of code files used during the code generation phase. The system manipulates these files in order to generate the code required by user. Commands store (Figure 12) contains all the voice command and the trigger handler classes which are executed when the user inputs that command.

find	*edu.ncstate.csc510.okeclipse.analyzeconsole*
inject	**edu.ncstate.csc510.okeclipse.injectcode**
main	*edu.ncstate.csc510.okeclipse.injectmain*
testing	**edu.ncstate.csc510.okeclipse.unittest**
status	*edu.ncstate.csc510.okeclipse.gitstatus*
commit	**edu.ncstate.csc510.okeclipse.gitcommit**
push	*edu.ncstate.csc510.okeclipse.gitpush*
pull	**edu.ncstate.csc510.okeclipse.gitpull**
convert	*edu.ncstate.csc510.okeclipse.convert*
generate	**edu.ncstate.csc510.okeclipse.generate**

Figure 12: Command library

The class diagram of all the relation classes and the dependency relation between them is shown in Figure 13.

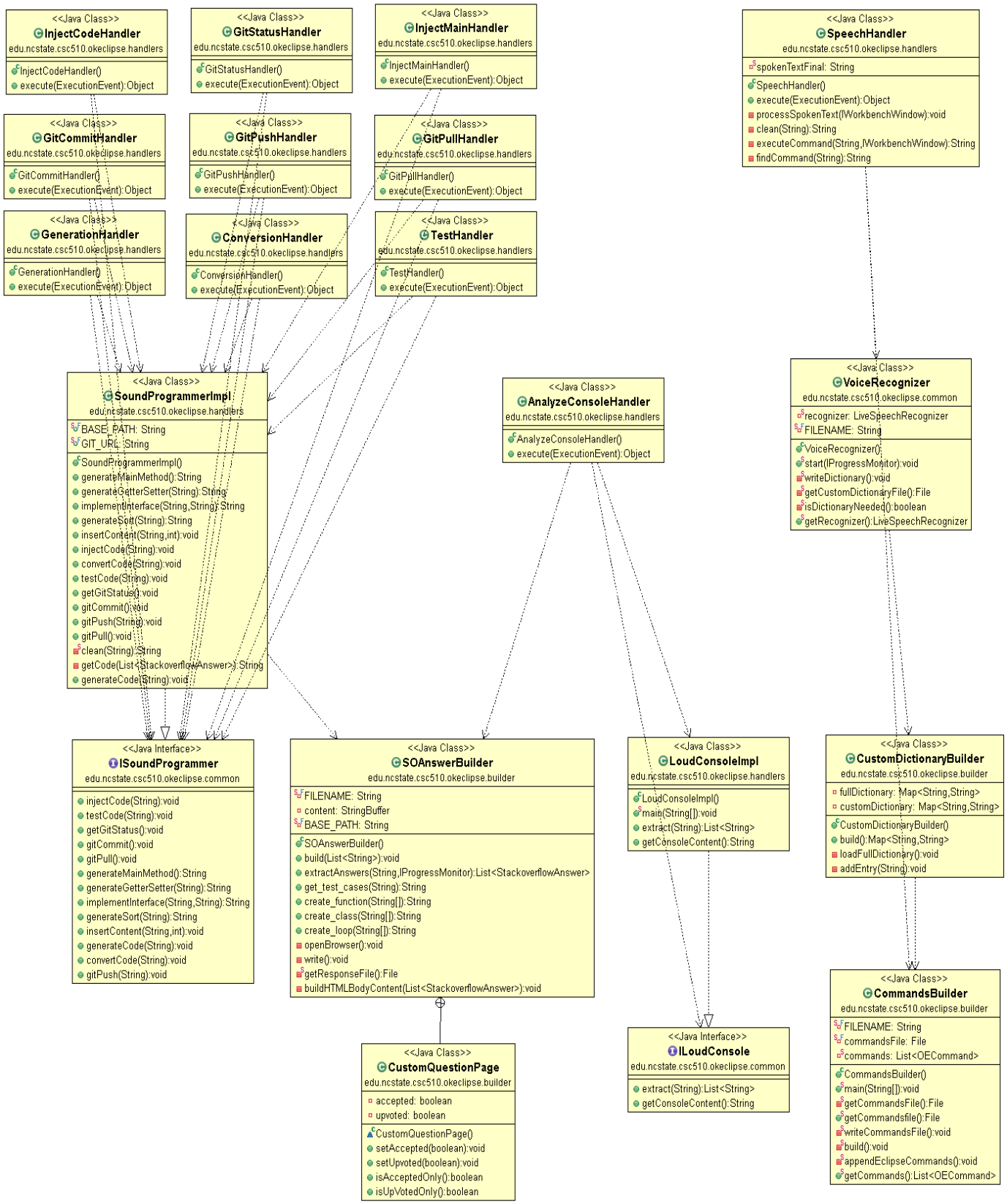

Figure 13: Class Diagram

6. EVALUATIONS

The following section provides information about the evaluation results and data of the internal and external evaluations performed by the team.

6.1 Internal Evaluations – Log Data

We collected log data on the usage of the plug-in in a CSV. The log data contained the following details:-

- Whether the voice command was correctly recognized or not?
- The percentage of words in the code generation sentence recognized.
- Time taken by different features until an output is rendered in Eclipse.

The voice recognition software Sphinx recognized ~70% of the voice commands. This number was high for modules other than code generation because it used a sentence. The percentage of the statement recognized for the code generation module were taken into consideration for the average amount i.e. if "generate * with four integer *" was recognized from "generate function with four integer parameters", then accuracy was noted to be 66.66%. Time taken is shown in Figure 14.

Feature	Time Taken
Code Generation	~0.7 secs
Code Conversion	~1.6 secs
Git Integration	~1.2 secs
Testing	~0.5 secs

Figure 14: Time analysis of functionalities

*The time taken by code conversion would depend on the speed of the network connection because it sends the file to the server for conversion and again sends the converted file back to the user.

*The same is the case with git integration. Time depends on the size of commit and the bandwidth speed.

6.2 External Evaluations – User Evaluations

This time we did not have a web based product, so we had to perform physical evaluation with the users for the system. The final evaluations were a blast. We received some great responses and feedback from our classmates and some hearty commendations as well. Our plug-in collects log data of the actions performed by the user on the plug-in, so we wanted to focus on the likes and dislikes and the satisfaction of the users in using the application.

The questions asked and the evaluation of responses is as follows:-

1. **Did the plugin work as expected and were you satisfied with the experience?**

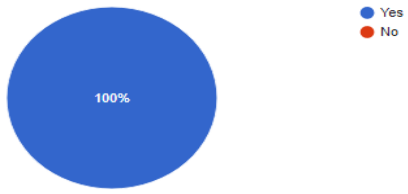

Figure 15 - Results for main plug-in evaluation

Reason - We asked this question for the sake of knowing whether there were some integration problems in the plug-in.

Results – As shown in Figure 15, all the users were satisfied with the experience and ran into no problems or errors.

2. **Was the time taken for code generation acceptable?**

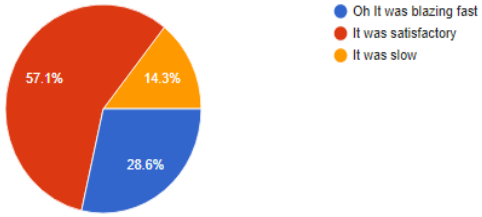

Figure 16 - Results for code generation evaluation

Reason - To know the public reaction as to whether it was slow for them or not and if it is what can be done about it?

Results - Most of the people found the generation to work fast (Figure 16). Now we tried to make it faster but it was a simple java program that parsed the input and generated the function so not much optimization could have been done about it.

3. **Was the time taken for push and pull acceptable?**

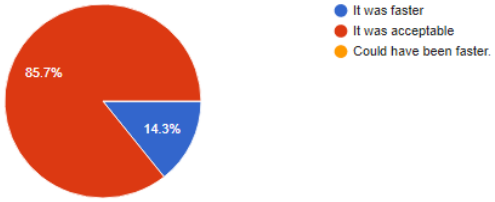

Figure 17 - Results for git integration evaluation

Reason - To know the public reaction as to whether it was slow for them or not and if it is what can be done about it?

Results - Most of the people found it satisfactory as seen in Figure 17. We time tested the Git library plug-in in the Eclipse and it performed around about similar to our plug-in; so no worries here.

4. **Did code generation work as you expect it to work and as it was described?**

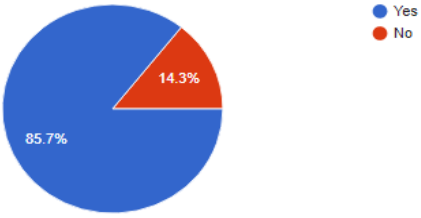

Figure 18 - Results for code generation evaluation

Reason - To understand whether the user's expectations regarding code generation were met!

Results - Most of the people were highly satisfied with the code generation module as seen in Figure 18.

5. **Were the results of the testing module correct and was the representation easy on the eye?**

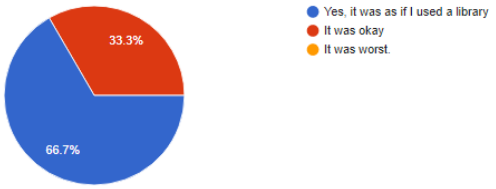

Figure 19 - Results for unit testing evaluation

Reason - To get a general view of whether people found the testing module easy to use and it produced correct results.

Results - All the reviews were positive and 67% (Figure 19) of them were really god saying that it was as if they used an inbuilt library.

6. **Did your converted code run or was a warning shown to you?**

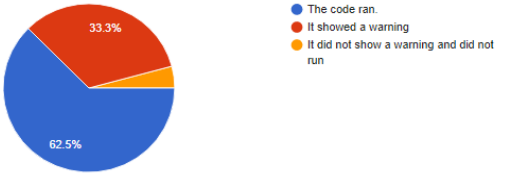

Figure 20 - Results for code conversion evaluation

Reason - To know whether the code converted was running or was a warning produced if it was not.

Results - Around 95% of the people found that it worked perfectly as in it showed a warning if it did not run or it run (Figure 20).

7. **Ratings for the full application and different features.**

The rating evaluations of Code conversion (Figure 21), Code generation (Figure 22), Unit testing (Figure 23), Git Integration (Figure 24) and the overall system (Figure 25) were collected.

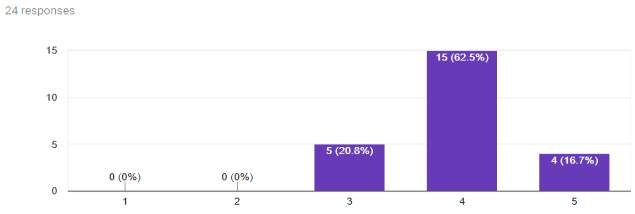

Figure 21 - Ratings for code conversion

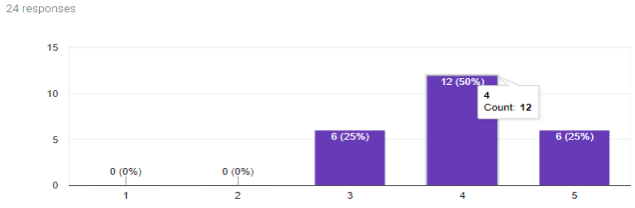

Figure 22 - Ratings for code generation

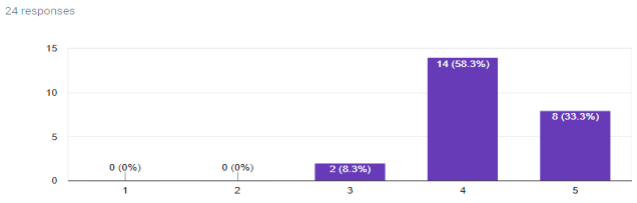

Figure 23 - Ratings for unit testing

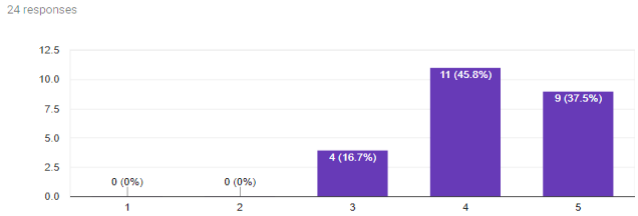

Figure 24 - Ratings for Git integration

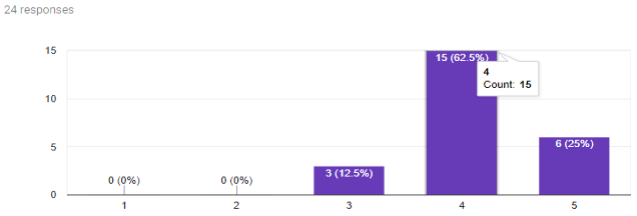

Figure 25 - Ratings for the overall plug-in.

7. CHALLENGES

We faced following challenges while developing the system:-

7.1 Constraint Dictionary

We have made use of the Sphinx system as our speech recognition software for this project. CMU Sphinx is a set of speech recognition development libraries and tools that can be linked into speech-enable applications. The large size of the Sphinx dictionary along with the open source nature of these libraries imposed certain limitations on the performance of the product as a whole. This caused us quite a few problems initially as several voice commands got misrecognized which resulted in the execution of a different action other than what the user had intended. After a significant amount of testing and experimenting with the Sphinx dictionary we came up with a way to circumvent this problem. We had concluded that the problem was due to the dictionary hosting a huge number of words that did not hold any relevance within the scope of our project and realized that these irrelevant words oftentimes caused the misrecognition of certain voice commands. Hence, we decided to create our own dictionary restricting its contents to only those words that fell within the scope of the eclipse.

Upon further testing, this “constrained dictionary” was found to perform more accurately while also reducing the processing time involved. As far as the performance in recognizing a full statement goes, Sphinx was not able to recognize a full sentence in normal form, a person had to stop for some 0.5 seconds between each word in a sentence, the Sphinx system took in the sentence as word by word. We had to tweak some changes for this to work as well. Although it worked, the performance was not as good as it we expected, the system was able to recognize only 50% of the words in a sentence. This was because the user had to

stop for 0.5 seconds between each word in a sentence and it was not very natural to do that and there were many similar words in a sentence. So it caused a lot of problems.

7.2 Code Conversion

The library called java2python, that we used for code conversion ran on Python and the ANTLR runtime of Python. We did not want the users to install Python and the ANTLR runtime for Python along with Eclipse for code conversion to run in our plug-in. We tried to install some mini Python environment where just the library could run on the local system, but it had some implementation challenges. We finally decided to do the whole conversion process on a server. We built the API for code conversion from scratch using Python and Flask.

7.3 Turnarounds

During our evaluations, some users found it a bit awkward when using the code generation feature because it's awkward requirement for speaking a sentence with 0.5 seconds stop between each word. They asked for some other options to do the same. Thus. for people who do not want to use the voice recognition systems, we developed different shortcuts for all the features; this way providing the application more options for usage.

7.4 Integration

Eclipse itself is a very heavy and clumsy environment for development. Integrating things with Eclipse caused a lot of time and trouble for us. We had developed a Python script which used Google Speech API voice recognition, but integrating it with Eclipse causes a lot of problems. One of the turnarounds was to call the Python script from command line and run that, but it was slow, had a weird command window pop up and sent the thread of execution from Eclipse to the command line. Even implementing the same thing in Java commanded for far too many dependencies, so we decided to stick with what we had and try to make it better in certain way.

8. FUTURE SCOPE

This iteration of development on OkEclipse opens up many pathways for development and improvements in the future. They are as follows:-

- The voice recognition could be certainly made more robust. Sphinx being an open-sourced library has some implementation limitations. Replacing it with Google [12] or Microsoft Speech API [11] might be better.
- Guaranteeing that the converted code in 100% correct and works is certainly another thing that can be done in the future. As well as including conversion from and to other languages might be added.
- The application currently generates only class structure, for loop structure and a subroutine structure. This can be extended to include complex classes and functions and other kinds of loops can also be added.
- The unit testing function works on all the basic primitive data types. But, user-defined data types can also be integrated to work with unit testing in the future.
- Other complex functions on the GitHub can be integrated as well.
- Other tests such as performance testing and integration testing could be included with some brainstorming.
- The implementation in this project is Eclipse specific. This can be done for other IDE's as well.
- Similarly, such a project can be made for different languages as well.

9. CONCLUSION

Thus, we have described the entire scope of the implementation of the project along with the software development methodologies that were adopted. The use cases guides for the future development of the project. The code for the project is fully modularized leaving scope for further development. The log data and user evaluations points at the success of the plug-in in making a programmer more productive during any part of the software development life cycle. Finally, we represent the challenges faced and future scopes as some possible extensions that could be added to the project in the future to make it more efficient and useful.

10. ACKNOWLEDGMENTS

Our special thanks to our Teaching Assistant Amritanshu Agrawal along with Prof. Timothy Menzies for helping us on ways to expand the current system and to develop a work plan to successfully execute the advances.

11. CHITS

QLF
TAS
XCY
PXP
YSG
NMG
PUQ
QLZ
RZL
IFP
KHJ
TLZ
NEN
API
TLH
YPG
PVX
HNK
QKP
QES
VUW
FUS
UOD
MGH

12. REFERENCES

[1] Prashant Yalla, Nakul Sharma. *Integrating Natural Language Processing and Software Engineering.* International Journal of Software Engineering and Its Applications, Vol. 9, No. 11 (2015), pp. 127-136 http://dx.doi.org/10.14257/ijseia.2015.9.11.12

[2] Imran Sarwar Bajwa, Ali Samad, Shahzad Mumtaz. *Object Oriented Software Modeling Using NLP Based Knowledge Extraction.* European Journal of Scientific Research ISSN 1450-216X Vol.35 No.1 (2009), pp 22-33 http://www.eurojournals.com/ejsr.htm

[3] Aman Chandra, Sakthivasan. *Enhancing SDLC traceability using NLP*. https://www.wipro.com/holmes/enhancing-sdlc-traceability-using-nlp/ [Online; accessed 3/21/2018]. 2018

[4] Eclipse. *Eclipse Plugin Development Environment*. http://www.eclipse.org/pde/. [Online; accessed 3/23/2018]. 2018

[5] GitHub Inc. *Software development platform · GitHub*. https://github.com. [Online; accessed 3/24/2018]. 2018

[6] Parameshachari B D, Sawan Kumar Gopy, Gooneshwaree Hurry, Tulsirai T. Gopaul. *A Study on Smart Home Control System through Speech*. International Journal of Computer Applications (0975 – 8887) Volume 69– No.19, May 2013

[7] CMU. *CMU Sphinx*. https://cmusphinx.github.io/wiki/. [Online; accessed 3/26/2018]. 2018.

[8] Agitar. *Agitar One.* http://www.agitar.com/solutions/products/agitarone.html . [Online; accessed 3/27/2018]. 2018.

[9] java2python. *Simple but effective tool to translate Java source code into Python*. https://pypi.python.org/pypi/java2python [Online; accessed 3/27/2018]. 2018.

[10] JGit. *The Eclipse Foundation* https://www.eclipse.org/jgit [Online; accessed 4/11/2018]. 2018.

[11] Bing Speech API. *Speech Recognition Software – Microsoft Azure* https://azure.microsoft.com/en-us/services/cognitive-services/speech/ [Online; accessed 4/22/2018]. 2018.

[12] Cloud Speech-to-Text. *Speech Recognition - Google Cloud* https://cloud.google.com/speech-to-text/ [Online; accessed 4/22/2018]. 2018.

[13] DigitalOcean. *Cloud Computing, Simplicity at Scale* https://www.digitalocean.com/ [Online; accessed 4/26/2018]. 2018.

[14] Flask. *A Python Microframework* http://flask.pocoo.org/ [Online; accessed 4/19/2018]. 2018.

[15] Python. https://www.python.org/ [Online; accessed 3/10/2018]. 2018.

[16] Java. https://java.com/en/ [Online; accessed 2/17/2018]. 2018.