Intelligent Code Editor

Design Document

Revised October 6, 2019

sdmay20-46
Ali Jannesari — Client & Adviser

Keaton Johnson — Systems Lead
Jonathan Novak — Machine Learning Lead
Matthew Orth — Meeting Facilitator
Garet Phelps — Report Manager
Isaac Spanier — Test Lead
John Jago — Software Lead

Team Email: sdmay20-46@iastate.edu
Team Website: https://sdmay20-46.sd.ece.iastate.edu

Executive Summary

Development Standards & Practices Used

Our work on the Intelligent Code Editor followed standard industry practices in software development. We followed agile practices with sprints lasting two weeks. We used test-driven development (TDD) when writing the software. In the software industry, engineering standards typically measure how well the software is tested, how well any manual processes are automated, and the quality of the software through the number of bugs or defects discovered. We will also follow these measures.

Summary of Requirements

Develop the functionality in an IDE or text editor to convert natural language (English) into Java code.

Applicable Courses from Iowa State University Curriculum

- COM S 227 (Introduction to Object-Oriented Programming)
- COM S 228 (Data Structures)
- COM S 309 (Software Project Management)
- COM S 319 (User Interfaces)
- COM S 472 (Artificial Intelligence)
- COM S 474 (Machine Learning)
- E E 526X (Deep Learning)

New skills/knowledge acquired that was not taught in courses

- Natural Language Processing

Table of Contents

1. Introduction

1.1 Acknowledgement

We would like to thank Professor Ali Jannesari for his guidance on this project not only as the primary client, but also as the faculty advisor. We would also like to thank Hung Phan for sharing his knowledge and for his involvement.

1.2 Project and Problem Statement

1.2.1 Problem Statement

With software becoming more prevalent in fields where it previously did not exist, more people must learn how to write programs to accomplish their work. One example is bioinformatics. At Iowa State, COM S 444: Bioinformatic Analysis is cross listed with Biology and Genetics, among other majors. Students in these majors do not necessarily have to become top programmers to do their work, but their work involves programming nonetheless. A lack of experience in programming can become an unnecessary hindrance to getting the work done, especially when trying to figure out the syntax of a programming language.

1.2.2 Solution Approach

Our goal is to provide a method for users to type what they want to accomplish in natural language and convert that natural language to the software code. This would allow someone who doesn't know how to write a particular statement in a programming language to type an approximation of that statement in English and have the editor convert the English to executable code.

The primary deliverable is a plugin to an existing integrated development environment (IDE) or text editor through which the user will interact with to translate natural language to code. Behind the scenes will be a classification engine that handles the actual conversion of English to code. Also produced will be a data set of possible natural language inputs that someone might type and the expected result, which will allow us to test the accuracy of our final product.

1.3 Operational Environment

The IDE plugin that we will develop along with the other software components will reside on computers and possibly remote servers. One major risk is that malicious software might be able to read data from our services. While we are not storing personal information, is it possible that people might not want their English (or the resulting code) exposed, as this could leak enough information for an attack on the program that the user is writing. Therefore, we will take into consideration the security of our software, especially if it communicates over a network.

1.4 Requirements

1.4.1 Functional Requirements

- User can select or otherwise input the text they wish to translate to code
- User can trigger a translate action
- The textual descriptions are replaced by the translated code fragments
- The translated code fragments can be executed as regular code
- Multiple selections can be translated at the same time

1.4.2 Non-Functional Requirements

- Translation time should be fast so it does not slow down the user's development pace
- The user interface should be clean and easy to understand

1.4.3 UI Requirements

- Translation action should be easily accessible from the text editor area

1.5 Intended Users and Uses

Users and their uses of this product:

- Someone who does not explicitly write code for their job, but needs to utilize programming for certain aspects and doesn't want to spend a large amount of time learning to code.
- A developer who is not intimately familiar with a particular language they would like to use, but can express the concepts they want in natural language or pseudo-code.

1.6 Assumptions and Limitations

1.6.1 Assumptions

- The user will enter their natural language statements in the general format we expect
- The user will enter their natural language statement in the location in the editor they want the translated code inserted
- The user will enter only natural language statements supported by the system's dataset

1.6.2 Limitations

- Translations will only be supported for common code syntax, methods, and classes.
- Translation may not generate the proper translation on the first attempt
- Translation will not occur for natural language statements not represented in the dataset
- Translations are only support from English to Java source code
- There is no budget provided for this project

1.7 Expected End Project and Deliverables

User Interface (November 2019):

The user interface (UI) will present the user with an Intelligent Development Environment (IDE) text editor where the user can enter their natural language statements representing the program they want to translate to code. The user interface will also allow the user to interact with the system to translate their natural language statements to equivalent code. Once the translate button is interacted with, the natural language statement will be passed to the classification/translation engine. The UI will then accept the translated code from the Translation Engine and display it in the editor at the location where the original natural language statement is.

Database/Dataset (December 2019):

The database will store the natural language statement representing a program and the expected code translation. The data from the database will serve as the training dataset for the classification model.

Classification and Translation Engine (February 2020):

The classification and translation engine will take the natural language statement passed from the UI. The statement will be passed as input to the classification model. The model will then use the input to produce an output that represents the tokenized expected code translation. This output will then be passed to the translation engine that will convert the tokenized code to the equivalent code. Finally, the translation engine will pass the expected translation to the UI.

Final Product (April 2020):

The final Intelligent Code Editor project will connect the User Interface, Database, and Classification/Translation Engine. This will create a complete end-to-end translation system.

2. Specifications and Analysis

2.1 Proposed Design

Our group is currently in the research and development stage of our project, so the below items are subject to change. The below items are the results of our initial research.

User Interface (UI):

An IntelliJ plugin will be created for the user interface. This plugin will present the user with an IDE text/code editor where they can enter their natural language statements. It will also give the user the ability to translate the natural language statement to code through some interaction button. The plugin will additionally connect to the classification and translation engine, passing the natural language statement and receiving and displaying the expected code translation. Finally, the translated code needs to be executable by the user.

Classification and Translation Engine:

The Classification engine will be implemented using OpenNMT-py. The OpenNMT-py model will be trained using the dataset stored in the database. The natural language statement from the UI will be input and the engine will output a tokenization of the expected translated code. This tokenized output will be passed to the translation engine, which will convert it to actual code that will be passed to the UI for display.

Database:

The database will be created using MongoDB. The database will store the natural language statements, expected translation pairs. MongoDB was chosen for the database as the natural language statements and expected translations will be unstructured data. The database will receive the dataset from the UI and will pass the dataset to the classification engine for training.

2.2 Design Analysis

Currently, our team is in the research and development stages. We have been researching tools and strategies to implement the user interface, database, and classification/translation engine. Through this research, initial strengths and weaknesses have been determined.

Strengths:

For the user interface, documentation for creating an IntelliJ plugin is significant and there is a large community for it. An IntelliJ plugin is also easy to install and use in the IntelliJ editor.

OpenNMT-py is a well-documented and widely used Neural Machine Translation engine that will convert the natural language statement to expected code. One major strength of OpenNMT-py is that it comes with training model templates, resembling modern techniques for neural machine translation. This will simplify the process of creating a model from scratch as that is something that requires a significant amount of experience.

Weaknesses:

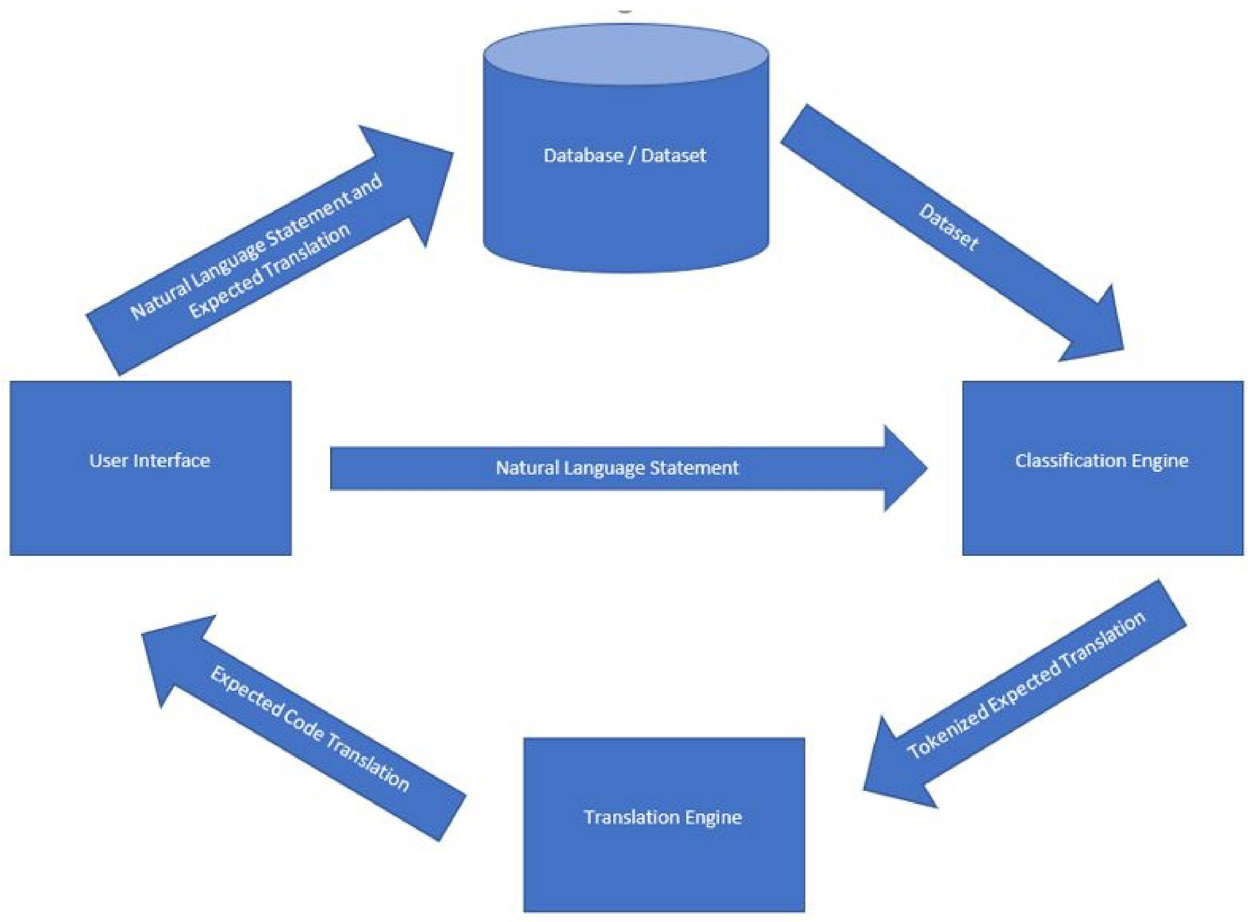
When creating an IntelliJ plugin, it does not allow the user to modify everything about the code editor, so we will have to ensure that we can modify what is necessary.

Some weaknesses of using OpenNMT-py are the models take a long time to train. However, this would be a problem regardless of the system we chose to implement it with. We will also need to determine how to create a representative dataset that will allow us to effectively train the model. This could take a long time to produce.

2.3 Development Process

Our team is following the Agile Development Process. Our team will have weekly meetings where we will review our project progress, update our task board, and plan future work. Bi-weekly meetings will be held where demos of our progress will be shown and feedback will be received to further improve our design. Testing our design throughout the design process will also be incorporated. This means that our requirements will be continuously changing throughout the development process.

2.4 Design Plan


Database / Dataset
Natural Language Statement and Expected Translation
Dataset
User Interface
Natural Language Statement
Classification Engine
Tokenized Expected Translation
Translation Engine
Expected Code Translation

The User Interface will pass the natural language statement to the classification engine. It will also pass the natural language statement and expected code translation to the Database if necessary.

The Database will store the natural language statement and expected code translation in the dataset. The Database will then pass the dataset to the Classification Engine for training.

The Classification Engine will train its model on the dataset. It will then take the natural language statement as input to the trained model. The trained model will then output the tokenized expected translation to the Translation Engine.

The Translation Engine will convert the tokenized input to the expected code translation that will be passed to the user interface to display and execute.

3. Statement of Work

3.1 Previous Work and Literature

User Interface:

Our research for the user interface started with the IntelliJ plugin API documentation [3]. Here we found useful information about what functionality was available for creating a custom IntelliJ plugins. We also utilized various documentation resources for creating an Eclipse Plugin [2] and Visual Studio Code Extension [1]; however, we did not find as many documentation resources and a large community for these plugins. This led us to choose an IntelliJ plugin.

Classification/Translation Engine:

We read many different research papers to determine methods to implement neural machine translation. From these, we determined that modern Neural Machine Translation mechanisms like the Transformer model are some of the best performing models [7]. With this information, we researched tools that can execute this kind of neural machine translation. We then found OpenNMT-py [4], which has easy interaction with modern neural machine translation models as we were able to train a model for language translation within a couple of hours.

Current systems that translate natural language to code require structured input from the user [6]. Our system will aim to allow the user to enter more unstructured natural language statements and convert that to equivalent code. Also, most other current systems only convert to a simpler language like the command line instead of a more complex programming language like our project will do.

Database:

We used our knowledge from CS 363 where we learned about MongoDB [5]. Since MongoDB works well with unstructured data, we determined this was an optimal database tool to use.

3.2 Technology Considerations

Strengths:

For the user interface, documentation for creating an IntelliJ plugin is abundant and there is a large community for it. An IntelliJ plugin is also easy to install and use in the IntelliJ editor, making is something a user would more likely use.

OpenNMT-py is a well-documented and widely used Neural Machine Translation engine that will convert our natural language statement to expected code. One major strength of OpenNMT-py is that is comes with training model templates resembling modern techniques for neural machine translation while also giving more customization options. This means we could use a model like the Transformer model without having to configure this ourselves. This will simplify the process of creating a model from scratch as that is something that requires a significant more amount of experience with Machine Learning and Natural Language Processing.

Weaknesses:

When creating an IntelliJ plugin, it does not allow the user to modify everything about the code editor, so we will have to ensure that we can modify what is necessary. This can be solved by either modifying our features or determining other mechanisms aside from IntelliJ plugins to implement those features.

Some weaknesses of using OpenNMT-py are the models take a long time to train. However, this would be a problem regardless of the system we chose to implement it with. This can be solved by requesting more powerful computers or servers to execute the training and classification of the model. Creating a representative dataset to effectively train the model will also be a challenge. This could take a long time to produce, but the process can be simplified by either utilizing already created datasets or determining other ways to automatically create the dataset.

3.3 Task Decomposition

The following project tasks are based on our initial research into the project, and may change.

- Research:
    - Research and test tools for creating IDE plugin / plugin
    - Research tools for natural language to code translation
    - Research tools for using a database to store a dataset

- User Interface (IntelliJ plugin):
    - Modify the IntelliJ IDE to allow the user to enter both natural language statements and code
    - Create mechanism that allows the user to translate the natural language statement to code
    - Connect the UI to the Database, Classification, and Translation Engine
- Database:
    - Create a dataset that will be used to train the classification model
    - Store the natural language statement, expected code translation pairs for training the training dataset
    - Pass the training dataset to the Classification Engine
- Classification Engine:
    - Use a dataset to train the natural language statement to code translation model
    - Be able to take a natural language statement from the UI as input and convert it to a tokenization of the expected code
    - Pass the expected code tokenization to the translation engine
- Translation Engine:
    - Take the expected code tokenization from the classification engine and translate to code
    - Pass the expected code translation to the UI
- Testing:
    - Write unit, GUI, and integrate tests
    - Run usability tests
    - Make improvements based on testing results

3.4 Possible Risks and Risk Management

A potential risk is a lack of knowledge in natural language processing. This starts with creating a large and representative enough database for training the classification model. Usually, datasets require thousands of entries, which could take a long time to create. For this, we may need to determine an automated way of creating the dataset or using an already generated dataset.

Once we have our dataset, training the classification model is a lengthy process, requiring powerful computing. We will likely need a dedicated computer or server for this purpose. Finally, once the model is trained, accuracy of the translation may be an issue. Even modern neural machine translation systems are unlikely to correctly translate natural language to code accurately [6]. We may need to give possible translation options to the user or require the user to enter more structured queries to compensate.

3.5 Project Proposed Milestones and Evaluation Criteria

The first milestone for the project will be researching and determining which user interface (UI), database, and classification/translation engine tools we want to use. This will uncover the initial direction of the project.

After the UI, database, and classification/translation tools are selected, the next milestone will be creating simple demos of using the UI, Database, and Classification/Translation engine for our purpose in isolation. This will involve giving fake data to these systems to ensure they behave correctly in isolation.

After everything is working in isolation, the next milestone will be to connect everything to ensure the system works end-to-end. This means the user will provide the natural language statement to the editor and the translation engine will provide the UI with the expected translated code to display.

Major milestones after this will involve improving the classification/translation engine to translate more complicated natural language statements (methods, classes, algorithms, etc.). This will likely require improving our training model with an updated dataset.

The final milestone for the project will involve testing and verification of the design. This will involve writing tests and running usability tests to ensure our design works as expected and can be used the way we expect.

3.6 Project Tracking Procedures

Our group will utilize GitLab, Trello, and GroupMe to track our progress throughout the semester. GitLab will be used to manage our project code. Each team member will develop in their individual branch and merge that into the master branch when ready.

Trello will be used to manage task creation and assignment. Each task will have a title, description, assigned member, and due date. There will be a backlog, doing, done, and completed column. These will represent tasks yet to be assigned, currently being worked on, done, and verified respectively.

GroupMe will be utilized for immediate communication with the team. Here, we will communicate meeting times, quick questions, and other communication.

3.7 Expected Results and Validation

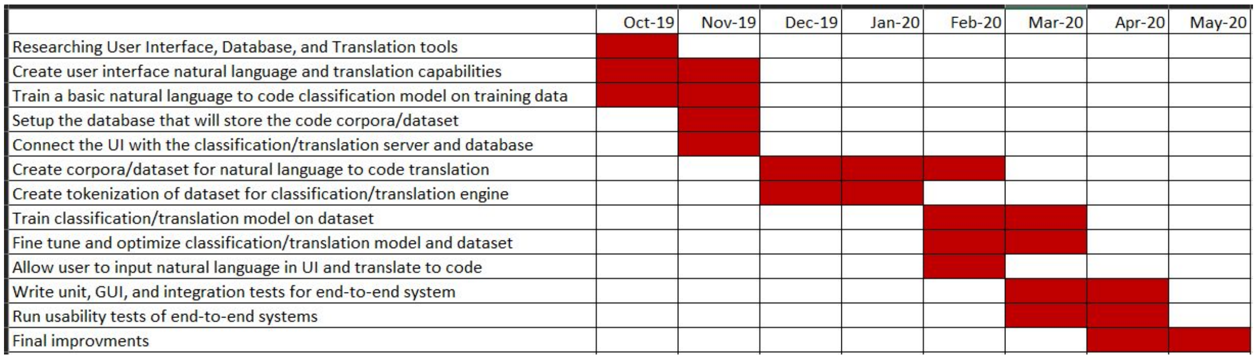
Our desired outcome is to create an end-to-end system where the user can enter a natural language statement into the code editor and the system will be able to translate the statement to equivalent code that will be displayed and be executable by the user in the editor.

Our implementation will be validated by creating unit and GUI test and through usability tests. This will ensure that our system behaves as expected and is easily usable by the users. This testing and usability tests will be developed throughout the development process to ensure we are creating an optimal solution.

4. Project Timeline, Estimated Resources, and Challenges

4.1 Project Timeline

This is a early-stage draft of our project timeline. It is subject to change.



	Oct-19	Nov-19	Dec-19	Jan-20	Feb-20	Mar-20	Apr-20	May-20
Researching User Interface, Database, and Translation tools	🟥							
Create user interface natural language and translation capabilities	🟥	🟥						
Train a basic natural language to code classification model on training data	🟥	🟥						
Setup the database that will store the code corpora/dataset		🟥						
Connect the UI with the classification/translation server and database		🟥						
Create corpora/dataset for natural language to code translation			🟥	🟥	🟥			
Create tokenization of dataset for classification/translation engine			🟥	🟥				
Train classification/translation model on dataset					🟥	🟥		
Fine tune and optimize classification/translation model and dataset					🟥	🟥		
Allow user to input natural language in UI and translate to code					🟥			
Write unit, GUI, and integration tests for end-to-end system						🟥	🟥	
Run usability tests of end-to-end systems						🟥	🟥	
Final improvments							🟥	🟥

Key:

- UI = User Interface
- GUI = Graphical User Interface

The first major stage of the project will be to conduct the initial research of the UI, database, and classification/translation engine tools we want to use.

The second major stage involves setting up and creating the basic functionality for the UI, database, and classification/translation engine. For the UI, the natural language input and translation interaction will be configured. The database will then be configured to store the dataset. Finally, the classification/translation engine will be configured, trained on a basic dataset, and setup as a server.

Connectivity between the UI, database, and classification/translation engine will be the next major task. This task will involve connecting the UI to the database and classification engine, the database to the classification/translation engine, and the classification/translation engine to the UI. We will then ensure that we can correctly send information between these systems.

The final stage will involve fine tuning, testing, and validating our results. This will start with fine tuning the dataset and classification/translation engine for best results. Next, we will begin writing the automated tests, end-to-end tests, and user validation testing. Finally, we will take the feedback from those tests to improve our final design.

4.2 Feasibility Assessment

The end-to-end natural language to code translation system explained above is very ambitious. Current systems struggle obtaining high accuracies for this purpose. Because of this, a more reasonable solution to this project may be to have the user enter more structured statements that the classification model can more easily learn from.

Another challenge will be creating a large enough dataset that will allow the trained model to be more accurate. As a result, our system may need to compensate by requiring the user to reenter their statement or give the user suggestions for translations. However, these restrictions and feasibility assessments will be more accurate as we progress more in the project.

4.3 Personnel Effort Requirements

These are very rough time estimates, and they will be updated to more accurate values once we begin the implementation phase of the project.

- Research: [100 hours]
    - Research and test tools for creating IDE plugin [30 hours]
    - Research tools for natural language to code translation [50 hours]
    - Research tools for using a database to store a dataset [20 hours]
- User Interface (IntelliJ plugin): [150 hours]
    - Modify the IntelliJ IDE to allow the user to enter both natural language statements and code [30 hours]
    - Create mechanism that allows the user to translate the natural language statement to code [20 hours]
    - Connect the UI to the Database, Classification, and Translation Engine [100 hours]
- Database: [200 hours]

    - Create a dataset that will be used to train the classification mode [125 hours]
    - Store the natural language statement, expected code translation pairs for training the training dataset [45 hours]
    - Pass the training dataset to the Classification Engine [30 hours]
- Classification Engine: [200 hours]
    - Use a dataset to train the natural language statement to code translation model [100 hours]
    - Be able to take a natural language statement from the UI as input and convert it to a tokenization of the expected code [50 hours]
    - Pass the expected code tokenization to the translation engine [50 hours]
- Translation Engine: [75 hours]
    - Take the expected code tokenization from the classification engine and translate to code [50 hours]
    - Pass the expected code translation to the UI [25 hours]
- Testing: [100 hours]
    - Write unit, GUI, and integrate tests [50 hours]
    - Run usability tests [20 hours]
    - Make improvements based on testing results [30 hours]

4.4 Other Resource Requirements

At this point, we envision requiring a dedicated server for training and use of our classification model as this process is computationally demanding. We will update this list as necessary throughout our development process.

4.5 Financial Requirements

This project does not provide or require any financial resources. All the required resources will be provided to us as needed.

5. Testing and Implementation

5.1 Interface Specifications

The chosen IDE for this project was the IntelliJ IDEA. This interface should be a good way to present our plugin to a user that is wanting to write code without using an actual language such as Java. IntelliJ's user interface is easy to become familiar with, but has many advanced options for more adept users. An IntelliJ plugin will be needed to integrate our software with the IDE.

As far as adding the interface to convert natural language to code with IntelliJ, there will be two options for a user to do this. The first will be selecting natural language that the user has typed directly into a file within IntelliJ and selecting our plugin to convert what they have written into code. The second interface will be an option to input natural language into a text box provided by the plugin, which when the user has finished, will generate the desired program based on the natural language provided.

5.2 Hardware and Software

No hardware will be provided other than the laptops and computers will be using to create the application plugin. For software we will be using IntelliJ IDEA, Eclipse, and Visual Studio Code to develop the application. GitLab will be used for source control, and keeping a clean code base. Trello will be used for project management and managing the amount of work each member is contributing.

5.3 Functional Testing

As mentioned in the feasibility section of this design document the success rate of similar projects is not high. However the functional tests requirements that we hope to see is that given the same natural language, or slight variations, the application should return the correct Java code with a 75% success rate. This is a high goal when compared with similar projects in the field, so this number is susceptible to change in the future.

The goal is to create a large enough data set in the coming months to be able to provide automated tests the data set and test the application throughout it's stages of development. This dataset should be wide enough to test the full functionality of the application and to be able to quickly spot any bugs or potential issues with application as it goes through its stages of production. The data set with be used to drive automated tests and should provide a baseline of how the function will work.

5.4 Non-Functional Testing

In terms of non-functional requirements for the application we need to ensure that functional requirements take precedence over these requirements. The first requirement we would like to focus on is the appearance of the plugin application. The plugin should be easy to use, and easy to access and should have great user interface that is fairly simple. This user interface should be essentially designed for a user who is adept with a computer but shouldn't overwhelm them with information.

5.5 Process

The Process of creating our tests relies heavily on creating or finding our dataset. We will need to find or create a variety of pseudocode phrases and examples that we can use to test our application. This is the first step, the second step is to create automated tests that take the dataset in and run the application with that input. Finally we need to compare the results of the tests with the expected autoput and refactor our application to ensure a higher success rate. Then the cycle repeats by adding more information to our dataset we can repeat the process and achieve a better product.

5.6 Results

Currently no results as we are still researching how to create our dataset for the tests.

6. Closing Material

6.1 Conclusion

6.1.1 Development Progress

So far in the development of the Intelligent Code Editor we have done the following:

- Looked into existing tools related to NLP to code
  - Discovered openNMT, an open source neural machine translation system. Used for research on various translations (image-to-text, English-to-Spanish), it could be a good resource for our project.
- Completed a literature analysis on research related to NLP and NLP to code
- Worked on the development of the plugin
  - Made a basic plugin for VScode, IntelliJ, and Eclipse
  - Ultimately decided to develop the plugin for IntelliJ
  - Plugins for different IDE's could be added later in development

6.1.2 Summary

The Goal for this project is to develop an application that assists the development of code by using natural language processing to convert normal text/pseudocode into functional code in an editor. To meet this end, we have settled on the following technologies:

- We will develop the plugin for IntelliJ. We settled on this due to IntelliJ's large amount of plugin documentation online, as well as it's focus on Java, which we are all familiar with.
- We will use OpenNMT.py to train a model for the NLP. OpenNMT.py is widely used and well documented, as well as being quoted as "production ready" by some companies, verifying its effectiveness.

6.2 References

[1] Code, V. (2019). *plugin API*. [online] Code.visualstudio.com. Available at: https://code.visualstudio.com/api [Accessed 24 Sep. 2019].

[2] Amsden, J. (2019). *Your First Plug-In*. [online] Eclipse.org. Available at: https://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html [Accessed 30 Sep. 2019].

[3] JetBrains IntelliJ Platform SDK. (2019). *Creating Your First Plugin*. [online] Available at: http://www.jetbrains.org/intellij/sdk/docs/basics/getting_started.html [Accessed 30 Sep. 2019].

[4] Opennmt.net. (2019). *Contents — OpenNMT-py documentation*. [online] Available at: http://opennmt.net/OpenNMT-py/ [Accessed 24 Sep. 2019].
[5] Docs.mongodb.com. (2019). *MongoDB Documentation*. [online] Available at: https://docs.mongodb.com/ [Accessed 24 Sep. 2019].
[6] Lin, X., Wang, C., Pang, D., Vu, K., Zettlemoyer, L. and Ernst, M. (2019). *Program Synthesis from Natural Language Using Recurrent Neural Networks*. [online] Seattle, WA, USA: Paul G. Allen School of Computer Science & Engineering. Available at: https://homes.cs.washington.edu/~mernst/pubs/nl-command-tr170301.pdf [Accessed 24 Sep. 2019].
[7] Wu, Y., Schuster, M., Chen, Z., Le, Q. and Norouzi, M. (2016). *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. [online] Google. Available at: https://arxiv.org/pdf/1609.08144.pdf [Accessed 30 Sep. 2019].

6.3 Appendices