Intelligent Code Editor

Design Document

Revised December 3, 2019

sdmay20-46
Professor 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 at the beginning; we switched to one week sprints as we became more familiar with the project and met more regularly. For the IntelliJ plugin, we practiced test-driven development (TDD). 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.

There are a couple specific IEEE standards that our team will informally adopt to assist in developing a quality product. The first is IEEE 1028-2008, which concerns software reviews and audits. Our technical reviews occur on GitLab whenever a merge request is submitted. Another is IEEE 16326-2009, which outlines the project management aspect of software development. This standard is implemented in the form of this design document. Additionally, IEEE 1008-1987 is implemented during the test planning and test implementation stages.

Summary of Requirements

The user will enter their natural language statement into the IDE. The natural language statement will be passed to a neural machine translation back-end system that will convert the natural language to equivalent Java code that will be displayed and executable by the user. A dataset will be created to train the neural machine translation model. This functionality will be developed in an Integrated Development Environment (IDE) or text editor to convert natural language (English) into Java code. Further optimizations will be made to the system in terms of learning method and translation accuracy.

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
- Dataset Creation
- IDE plugin development

Table of Contents

List of Figures

List of Tables

Definitions:

- UI = User Interface
- GUI = Graphical User Interface

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 their 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 the natural language to code. Behind the scenes will be a classification/translation engine that handles the actual conversion of English to code using a trained model. Also produced will be a dataset of possible natural language inputs that someone might type and the expected code translations, which will allow us to train our model and test the accuracy of our final product.

1.3 Operational Environment

The IDE plugin 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 or release of confidential information. 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 model should learn using an unsupervised method

1.4.3 UI Requirements

- Translation action should be easily accessible from the text editor area
- The user interface should be clean and easy to understand

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 does not 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.
- Integration with speech-to-text technology to allow programming by dictation.

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.
- The translation may not generate the proper translation on the first attempt or at all for certain inputs
- Translation will not occur for natural language statements not represented in the dataset
- Translations are only supported from English to Java source code

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.

Classification and Translation Engine (December 2019):

The classification engine will be trained from a dataset of natural language, expected code translation pairs. This model (along with the natural language statement from the UI) will be passed to the translation engine. The model will then use the input to produce an output of the expected code translation. Finally, the translation engine will pass the expected translation to the UI.

Dataset (February 2020):

The dataset will contain the natural language statement, expected code translation pairs. This dataset will contain a source (natural language statements) and target (expected code translation) files for training, validation, and testing. Experimentation will be done on the effectiveness between manually and automatically creating a dataset as well as using already created datasets. The dataset will be created locally and will be stored on the translation server.

Model and Dataset Optimization and Self-Learning (March 2020):

Throughout the development process, the training model (architecture and hyperparameters) and dataset will be optimized to achieve the best performance. The performance will be measured through neural machine translation metrics, research, and observed translation results. In addition, we will configure the classification and translation model to be unsupervised and self-learning. This will allow the system to learn from the input the user provides instead of labeled data from a dataset, creating a better performing system over time.

Final Product (April 2020):

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

2. Specifications and Analysis

2.1 Proposed Design

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 will be executable by the user.

Classification and Translation Engine:

The Classification and Translation engine will be implemented using OpenNMT-py. The OpenNMT-py model will be trained using a dataset containing natural language statement, expected code translation pairs. Future plans will allow OpenNMT-py to support unsupervised learning on the input specified by the user. The natural language statement from the UI will be input and the engine will output the expected translated code to the UI.

Dataset:

The dataset will contain the natural language statement, expected code translation pairs. This dataset will contain a source (natural language statements) and target (expected code translation) files for training, validation, and testing. Experimentation will be done on the effectiveness between manually and automatically creating a dataset as well as using already created datasets. In addition, the dataset will contain research on synonyms, sentence permutations, and various English dialects. The dataset will be created locally and will be stored on the translation server.

2.2 Design Analysis

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, making it easier and more likely for a user to utilize.

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

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 for our requirements.

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. Allowing OpenNMT-py to support unsupervised learning will help with this issue, but 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. These meetings will ensure there is clear communication between our group and our client. Testing our design throughout the design process will also be incorporated to validate our design. This means that our requirements will be continuously changing and improved throughout the development process.

2.4 Design Plan

The below diagram shows our design plan:

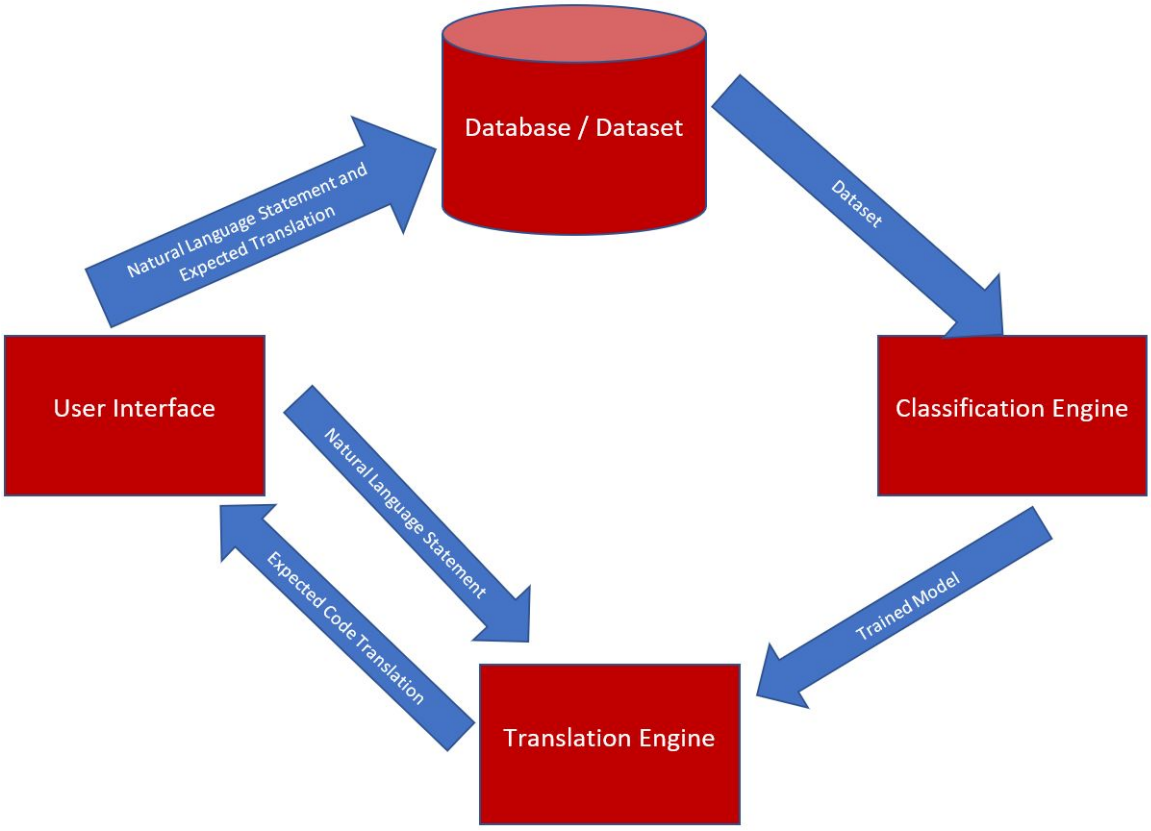

Figure 1. Detailed Design Diagram

User Interface:

The user will first enter their natural language statement into the code editor. Then after the user clicks the translate button, the User Interface will pass the natural language statement to the translation engine. The user (or developer) will also pass the natural language statement, expected code translation pairs to the dataset. The user interface implements many of the functional requirements.

Dataset:

The Dataset will store the natural language statement and expected code translation in the respective source (natural language) and target (expected code translation) files that are split into training, validation, and test sets. These dataset files will then be passed to the Classification Engine for training.

Classification Engine:

The Classification Engine will train the model on the provided dataset, using the natural language statement as the input and the expected code translation as the output. After training has been completed, the model will calculate translation metrics on the testing dataset to determine effectiveness of results. The Classification Engine will then pass the trained model to the translation engine to use for translation.

Translation Engine:

The Translation Engine will take the trained model from the Classification Engine and the input from the User Interface to generate the expected code translation. This expected code translation will be passed to the user interface for display. Eventually, the translation engine will also support unsupervised learning on the natural language input provided by the user interface. The dataset, classification, and translation engine satisfy many of the non-functional requirements.

Architecture Diagram:

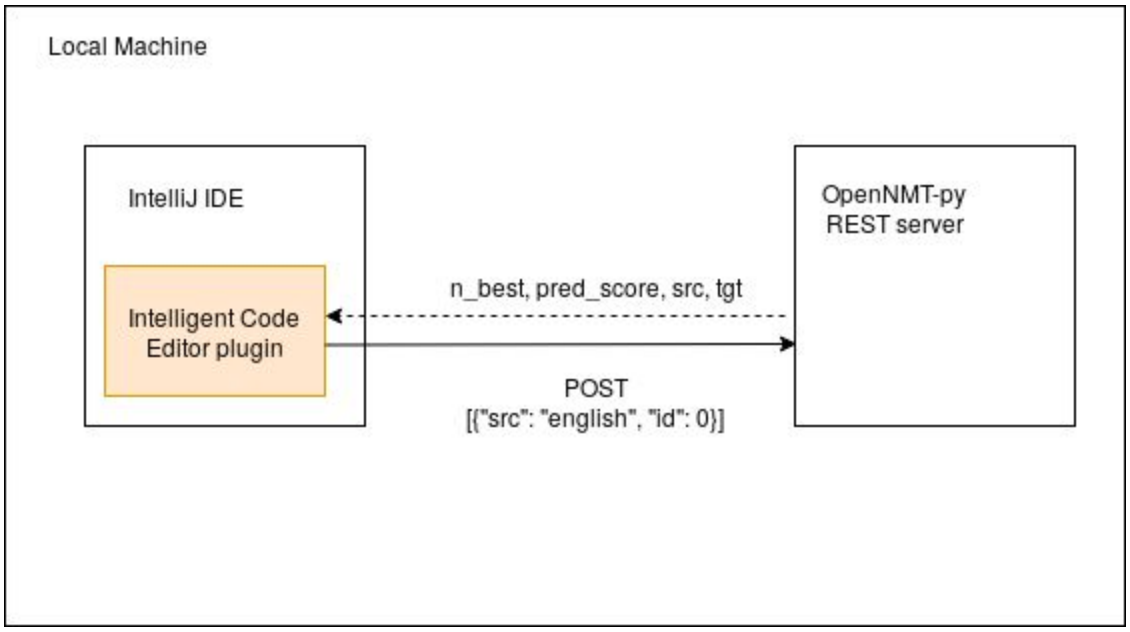

Figure 2. User Interface and Translation Server Architecture Diagram

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 [1]. Here we found useful information about what functionality was available for creating a custom IntelliJ plugin. We also utilized various documentation resources for creating an Eclipse Plugin [2] and Visual Studio Code Extension [3]; 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 papers, we determined that modern Neural Machine Translation mechanisms like the Transformer model are some of the best performing models [4]. With this information, we researched tools that can execute this kind of neural machine translation. We then found OpenNMT-py [5], which has easy interaction with modern neural machine translation models as we were able to train a basic model for language translation within a couple of hours.

Current systems that translate the 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. This will allow more individuals to use our system. Additionally, current systems require a supervised learning method while our system will aim to allow our model to be trained in an unsupervised manner.

Dataset:

The dataset will be modeled in the way OpenNMT-py expects: a source (natural language statements) and target (expected code translation) files for training, validation, and testing.

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 the expected code. One major strength of OpenNMT-py is that it 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 and original research 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. Allowing OpenNMT-py to support unsupervised learning will also improve this process. Additionally, OpenNMT-py does not support all neural machine translation architectures.

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/extension
    - Research tools for natural language to code translation

    - Research tools for creating a dataset
- User Interface (IntelliJ plugin):
    - Modify the IntelliJ IDE to allow the user to enter both natural language statements and code
    - Create a mechanism that allows the user to translate the natural language statement to code
    - Consider the code context to translate variables, methods, and class names to generic names when passing into OpenNMT-py
    - Connect the UI to the Dataset, Classification, and Translation Engine
- Dataset:
    - Research word synonyms, word permutations, and English dialects
    - Create or find a dataset that will be used to train the classification model
    - Store the natural language statement, expected code translation pairs for the source and target training, validation, and testing dataset
    - Optimize the dataset for best performance
    - Pass the training dataset to the Classification Engine
- Classification Engine:
    - Use a dataset to train the natural language statement to expected code translation model
    - Optimize the model architecture and hyperparameters for best performance
    - Pass the trained model to the Translation Engine
- Translation Engine:
    - Take as input the trained model from the Classification Engine and the natural language input from the User Interface
    - Enable the translation engine to support unsupervised learning based on the input provided by the user interface
    - Pass the expected code translation to the UI
- Testing:
    - Write unit, GUI, and integration 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 dataset 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 need a dedicated computer or server for this purpose. Finally, once the model is trained, the accuracy of the translation may be an issue. Even modern neural machine translation systems are unlikely to correctly translate the 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), dataset, and classification/translation engine tools we want to use and create. This will uncover the initial direction of the project.

After the UI, dataset, and classification/translation tools are selected, the next milestone will be creating simple demos of using the UI, Dataset, 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. The UI should allow the user to enter natural language statements and be able to convert them to code using a fake back-end system. The classification/translation engine should take a pre-made dataset containing natural language statements and equivalent code to train and test the model. The model translations should have about 50% accuracy.

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 dataset and classification/translation engine to translate more complicated natural language statements (methods, classes, algorithms, etc.). Additional work will be done to allow the classification/translation engine to support unsupervised learning based on the input provided by the user. This will likely require improving our training model with an updated dataset. Once completed, the classification/translation engine will meet our accuracy requirements and support self-learning when a user enters a natural language statement through the system.

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. Successful completion will have all required functionality tested, verified, and accepted.

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. One member will be in charge of merging the branches together.

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. It is expected each member accomplishes their tasks assigned for each sprint.

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 tests 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.

In addition, our training model and dataset's performance will be validated using neural machine translation performance measures, research, and observation of translation.

4. Project Timeline, Estimated Resources, and Challenges

4.1 Project Timeline

Table I

Intelligent Code Editor Project Timeline

	Oct-19	Nov-19	Dec-19	Jan-20	Feb-20	Mar-20	Apr-20	May-20
Research User Interface, Database, and Translation tools								
Create user interface natural language input and translation capabilities								
Train a basic natural language to code classification model on pre-made dataset								
Optimize neural machine translation architectures and hyperparameters								
Create corpora/dataset for natural language to code translation								
Train classification/translation model on dataset								
Connect the UI with the classification/translation server and database								
Update and optimize classification/translation model and dataset								
Allow user to input natural language in UI and translate to code (accurate end-to-end integration)								
Enable classification/translation engine to support unsupervised learning								
Write unit, GUI, and integration tests for end-to-end system								
Run usability tests of end-to-end systems								
Final improvments								

The first major stage of the project was to conduct the initial research of the UI, dataset, and classification/translation engine tools we want to use, which was completed in late October. This research set the foundation for the remainder of the project.

The second major stage involves setting up and creating the basic functionality for the UI, dataset, and classification/translation engine. For the UI, the natural language input and translation interaction were configured. The classification/translation engine was configured, trained on a basic dataset, and set up as a server. Finally, we created a targeted Java print dataset. This work was completed late November, and it will be used to confirm our design for the remainder of the project.

Connectivity between the UI, dataset, and classification/translation engine will be the next major task. This task will involve connecting the UI to the dataset and classification engine, the dataset 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. This work will be completed once we obtain access to a dedicated server in February/March.

The final stage will involve optimizing, testing, and validating our results. This will start with fine-tuning the dataset and classification/translation engine for the best results. We

will also enable OpenNMT-py to support unsupervised learning based on the input provided by the user. These steps will create a more accurate translation system. Then we will continue 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. Input preprocessing before passing the natural language input into the translation engine could also improve our accuracy. Additionally, implementing the model learning method as unsupervised learning could help the system learn as more data is entered by the user.

Implementing the learning method as unsupervised learning is also a significant challenge as current neural machine translation systems utilize supervised learning. Research and learning will be done by our team to determine the method and ability to implement this feature.

Another challenge will be creating a large and representative 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. Additionally, we may need to utilize already made datasets or determine an automatic method for dataset generation. 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.

Table II

Intelligent Code Editor Personal Effort Requirements

Research	**75 hours**
Research IDE Plugin/Extension	25 hours
Research Natural Language to Code Translation	25 hours
Research Creating Dataset	25 hours
User Interface (IntelliJ Plugin)	**60 hours**
Enter Natural Language Statements and Code Functionality	10 hours
Create Mechanism to Translate Natural Language to Code	10 hours
Consider Code Context Before Passing Natural Language to OpenNMT-py	35 hours
Connect the UI to the Dataset, Classification, and Translation Engine	5 hours
Dataset	**175 hours**
Research Word Synonyms, Word Permutations, and Different English Dialects	25 hours
Create or Find a Dataset That Will Be Used To Train The Classification Model	100 hours
Store Natural Language Statement, Expected Code Translation Pairs in Dataset	20 hours
Optimize Dataset for Optimal Performance	25 hours
Pass the Training Dataset to the Classification Engine	5 hours
Classification Engine	**80 hours**
Use Dataset to Train The Natural Language Statement to Expected Code Translation Model	25 hours
Optimize Model Architecture and Hyperparameters for Best Performance	50 hours
Pass the Training Model to the Translation Engine	5 hours

Translation Engine	**115 hours**
Receive Input and Trained Model from User Interface and Classification Engine	5 hours
Enable Translation Engine to Support Unsupervised Learning	100 hours
Pass the Expected Code Translation to the UI	10 hours
Testing	**100 hours**
Write Unit, GUI, and Integration Tests	25 hours
Run usability Tests	25 hours
Make Improvements Based on Testing Results	50 hours

4.4 Other Resource Requirements

A dedicated GPU server is required and provided to train our classification model on the dataset. Our group has been given access to the Pronto GPU server at Iowa State University [7]. This server is a shared computing resource that allows access to powerful computing resources, including multiple GPUs. This will speed up our training and development process as well as allow us to train models with more complex architectures.

Additionally, a dedicated server will be provided to run our classification/translation engine's REST server. This server will connect with the front-end to allow data to be sent from the user interface to the classification/translation server.

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

This project does not contain any interfacing between hardware and software components, as it consists entirely of three software parts: the IntelliJ plugin (client), OpenNMT-py (the translation engine into which we feed our model), and the OpenNMT-py REST server. However, there are interfaces between the user and the plugin as well as the plugin and the OpenNMT-py system, which are described below.

The chosen IDE for this project is an IntelliJ plugin. This interface is a good way to present our plugin to a user that is wanting to write code without using an actual programming 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.

For adding an interface to convert the natural language to code with IntelliJ, the user will select the natural language they typed directly into a file within IntelliJ and select our plugin to convert what they have written into code. This process will interface with the end-to-end system that consists of the dataset and classification/translation REST engine, which is located on a dedicated server.

5.2 Hardware and Software

For hardware, the Pronto shared computing resource will be used to train and test the classification model, and we will use our personal computers for the rest of the development, which includes creating the user interface. In order to provide our translation engine as a service, we will host it on shared server hardware at the university.

In terms of software, we will use the IntelliJ Platform SDK to develop the IntelliJ plugin that will be the user interface for our target audience. OpenNMT-py will be used to implement the classification/translation engine and server. GitLab will be used for source control, collaboration, and maintaining the project history. Trello will be used for project management and distributing the work between each team member.

5.3 Functional Testing

For Functional Testing of our project, we want to provide a natural language to Java language translation. So far, we have focused on the Java print statements and Java method invocations. We will write automated unit, GUI, and integration tests for the user interface component of the project. This ensures that the system both meets our specified requirements and behaves correctly as an end-to-end system. These tests will be written throughout the development process to catch bugs early, and where appropriate, the test-driven development methodology will be used.

Our client and graduate student will perform acceptance testing on the plugin at weekly meetings to evaluate the direction our software is going, specifically the user interface aspect, to confirm that the product meets requirements, and to make recommendations for future improvements.

Following standard Java practices, the Intellij plugin has tests located in src/test/java. These tests cover functionality within the plugin itself, such as whether the translate action calls an external API and whether text preprocessing is performed correctly.

5.4 Non-Functional Testing

As mentioned in the feasibility section of this design document, the success rate of similar projects is not high. Our application aims to return the correct and executable Java code with a 50-70% 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. Most of the current systems obtain around a 25%-30% success rate.

In terms of other non-functional requirements for the application, 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 a user interface that is fairly simple and fast. This user interface should be essentially designed for a user who is adept with a computer but shouldn't overwhelm them with information. To evaluate our success in making an intuitive interface, at some point we will have a non-technical user attempt to enter English "code" into the editor and note if they are able to complete their task without struggling.

Another area of non-functional testing that we will consider is security. Given that we will have a public REST server running to serve clients the results of our translated model, we will ensure that best practices in security are followed, such as blocking unused ports from the outside internet and delivering data over secure TLS connections. Additional group members will verify that the setup is indeed secure.

5.5 Process

Unit, GUI, and integration tests will be written throughout the development process to mainly test the functionality of the user interface and the end-to-end system. This will ensure that we catch bugs or missed requirements early. During the development of our dataset and classification/translation engine, we will test the effectiveness of our dataset and classification/translation engine architecture and hyperparameters. This will be done by running our dataset through our designed neural machine translation architecture and observing the results according to research and neural machine translation metrics such a BLEU and calculating the translation accuracy. Modifications to our system will be made where unexpected results are observed. This cycle will continue until we achieve all requirements and achieve accurate translation results.

5.6 Results

Dataset Creation:

First, we generated a simple Java print statement dataset that contained different ways to say print with some different values to print. The translated results can be seen below:

```
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("");
```

Figure 3. Initial Java Print Dataset Output

We observed that System.out.println(“”); was being correctly generated for each entry; however, the value that was supposed to be printed was not. From this testing, we learned

that the default OpenNMT-py configuration can correctly translate common code (in this case the Java print statement).

We then created a more targeted Java print statement dataset consisting of "print number" where number represents all numbers from 1 to 100,000 written in words and in numerical representation. After running this new dataset through our translation model, we achieved the following results:

```
System.out.println("seventy-six thousand and seventy-three");
System.out.println(25221);
System.out.println(19933);
System.out.println("four thousand and four");
System.out.println(19933);
System.out.println(25820);
System.out.println(19933);
System.out.println(19933);
System.out.println("thirty-two thousand and forty-nine");
System.out.println(19933);
System.out.println(25820);
```

Figure 4. Java Print Number Output

We could see that the model is now populating the print statement with values; however, these values were not correct (they were supposed to be numbers 1-10). After observing these results, we started experimenting with the OpenNMT-py network architecture and hyperparameters.

OpenNMT-py Configuration:

First, we tried using the Transformer architecture. Passing the same dataset through, we achieved the following results:

```
hundred");
print
three");
four");
print
six hundred");
System.out.println("seven hundred and ten");
hundred");
print
print
twenty-two");
4302
```

Figure 5. Java Print Number Transformer Dataset

For some reason, the results achieved were worse than the basic network architecture. This could have been due to the Transformer model requiring many GPU resources, so we have to scale down some of the parameters.

Next, we tried using an RNN network architecture. After running the same dataset through the RNN architecture, we achieved the following results:

```
System.out.println("one");
System.out.println("two");
System.out.println("three");
System.out.println("four");
System.out.println("five");
System.out.println("six");
System.out.println("seven");
System.out.println("eight");
System.out.println("nine");
System.out.println("ten");
```

Figure 6. Java Print Number RNN Output

These results are exactly what we expect. From this experiment, we learned that values that are contained within the training dataset have a high probability of being translated correctly during testing.

Complex Dataset:

After confirming our results using a basic dataset, we then tested our results on more complex datasets. The first dataset we tried was the Python Parallel Code Corpora [8]. This dataset consisted of automatically generated natural language to code translations

generated from Stack Overflow. After running this dataset through OpenNMT-py, we achieved the following results:

```
DCSP return DCNL
DCSP return none'
DCSP return 'Return
DCSP return it.
DCSP return 'Prepare
DCSP 'Test
DCSP return DCNL
DCSP return 'Output
DCSP return first
DCSP return option
DCSP return for
DCSP return #1463'
DCSP return True.
```

Figure 7. Python Parallel Corpora Output

The results achieved for this dataset are not ideal. It did pull out common syntax (DCSP and return), but the results were not accurate. This is likely due to the natural language statements and expected code translation being very complex (many lines with not common syntax).

We then began researching other datasets when we found the Conala dataset [9]. This dataset again was created using automated mining methods from Stack Overflow. However, this dataset also contained manually generated data points that the automatic generation system used when mining. After running this dataset through our system with the Transformer architecture on more powerful GPU resources, we achieved the following results:

```
connection . send ( 'HTTP/1.0#SPACE#200#SPACE#OK\r\n\r\n' ) #NEWLINE#
print ( content . decode ( 'utf8' ) ) #NEWLINE#
len ( set ( mylist ) ) == 1 #NEWLINE#
'hello#SPACE#there#SPACE#%(5)s' % { '5' : 'you' } #NEWLINE#
indices = [ i for i , x in enumerate ( my_list ) if x == 'whatever' ] #NEWLINE#
results_union = set ( ) . union ( * results_list ) #NEWLINE#
results_union = set ( ) . union ( * results_list ) #NEWLINE#
strings . sort ( key = lambda x : x . resultType ) #NEWLINE#
[ list ( x ) for x in zip ( * sorted ( zip ( list1 , list2 ) , key = lambda pair : pair [ 0 ] ) ) ] #NEWLINE#
```

Figure 8. Conala Dataset Output

We observed that the results resembled their expected code much more closely. This led us to determine how we can create a more accurate dataset that would achieve greater performance or how we can update our network architecture to achieve better results on the Conala dataset.

Next, we generated a dataset targeting the Java print statement. Our dataset contained various hardcoded strings, arithmetic, variables, and function calls. After running the dataset through our system, we received the following results and metrics:

```
SENT 128: ['write', 'current', 'thread']
PRED 128: System.out.println ( Thread.currentThread() ) ;
PRED SCORE: -0.6242
GOLD 128: System.out.println ( Thread.currentThread() ) ;
GOLD SCORE: -0.6242

SENT 129: ['output', 'what?!', 'to', 'console']
PRED 129: System.out.println ( "Oaptain Scott. We" ) ;
PRED SCORE: -6.0800
GOLD 129: System.out.println ( <unk> ) ;
GOLD SCORE: -15.9697

SENT 130: ['write', 'are', 'cut', 'flowers', 'with', 'to', 'console']
PRED 130: System.out.println ( "are dying. It's the last" ) ;
PRED SCORE: -1.6209
GOLD 130: System.out.println ( "are <unk> flowers with" ) ;
GOLD SCORE: -29.0879

SENT 131: ['output', 'A', 'couple']
PRED 131: System.out.println ( "A couple" ) ;
PRED SCORE: -1.5632
GOLD 131: System.out.println ( "A couple" ) ;
GOLD SCORE: -1.5632

SENT 132: ['output', 'York.', 'It', 'looks', 'like', "we'll", 'to', 'console']
PRED 132: System.out.println ( "It felt like about" ) ;
PRED SCORE: -0.9181
GOLD 132: System.out.println ( <unk> It <unk> like <unk> ) ;
GOLD SCORE: -50.2045

SENT 133: ['output', 'radio.']
PRED 133:
PRED SCORE: -5.8935
GOLD 133: System.out.println ( <unk> ) ;
GOLD SCORE: -22.7214

SENT 134: ['print', 'out', '465', '/', '124']
PRED 134: System.out.println( 3 );
PRED SCORE: -0.4165
GOLD 134: System.out.println( 3 );
GOLD SCORE: -0.4165
```

Figure 9. Conala Dataset Translations

```
BLEU = 31.61, 67.2/57.1/37.5/21.4 (BP=0.755, ratio=0.780, hyp_len=2079, ref_len=2664)
```

Figure 10. Conala Dataset BLEU Score

Accuracy = 40%

We observed that the arithmetic statements were translated correctly almost all the time, but string values were not. Additionally, variables and function calls had an accurate translation. These results led us to create a dataset that supports more generalized natural language inputs.

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[4].
- Completed a literature analysis on research related to NLP (natural language processing) 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
- Created a basic IntelliJ plugin:
    - Allows the user to enter and select natural language in the code editor
    - Passes the entered natural language statement to the translation engine
    - Receives the expected code translation back from the translation engine
    - Uses context to assign variables a generic name myVar to ease the burden on the translation engine
- OpenNMT-py Configuration:
    - Configured OpenNMT-py on the Pronto GPU server
    - Determined how to run a dataset through the classification engine
    - Determine how to adjust network architecture and hyperparameters
    - Researched and tested different datasets, architectures, and hyperparameters
- Dataset Creation:
    - Created a Java print statement dataset to use as a baseline for our translation results

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

We now have three main deliverables that we will be working on:

- The baseline for the IntelliJ project is done. From here improvements will be made in gathering context from the surrounding code to make the sent data easier for the OpenNMT-py to interpret.
- The OpenNMT-py will continue to be optimized to better translate our natural language.
- A Java dataset is being continuously developed that the model will be trained on. Right now we are only working on the println method.

6.2 References

[1] 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].

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

[4] 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].

[5] Opennmt.net. (2019). *Contents — OpenNMT-py documentation*. [online] Available at: http://opennmt.net/OpenNMT-py/ [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] Researchit.las.iastate.edu. (2019). *Pronto Job Manager | ResearchIT*. [online] Available at: https://researchit.las.iastate.edu/pronto [Accessed 24 Oct. 2019].
[8] Valerio, A., Barone, M. and Sennrich, R. (2017). A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. [online] Available at: https://arxiv.org/pdf/1707.02275.pdf [Accessed 24 Oct. 2019].
[9] Yin, P., Deng, B., Chen, E., Vasilescu, B. and Neubig, G. (2018). Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. [online] Available at: https://arxiv.org/pdf/1805.08949.pdf [Accessed 24 Oct. 2019].

6.3 Appendices

We do not have any appendices at this time.