**Running Commentary**

**Introduction**

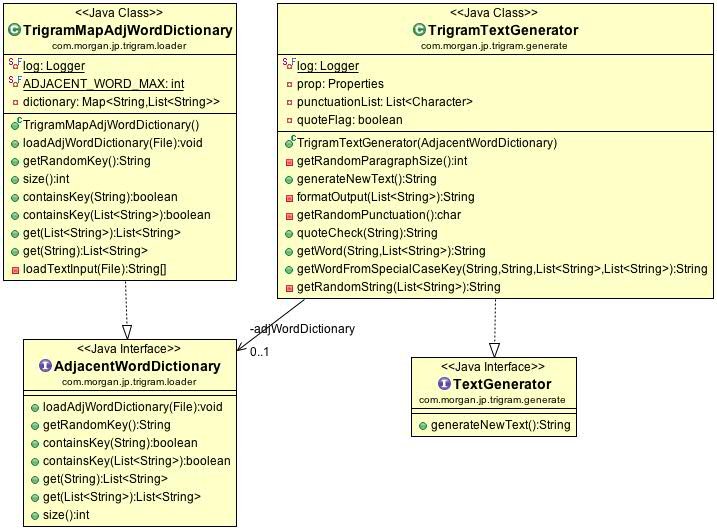
This document contains the ongoing running commentary and design decisions to create the Trigram Kata program. Demo and install videos of the program can also be found at: [www.tony-tran.com/trigram](http://www.tony-tran.com/trigram)

OR the source code and artifacts can be found at: <https://github.com/toant13/TrigramKata>

**Class layout**

My goal was to architect the classes in such a way that they were as compartmentalized as possible. I knew that kata required some kind of dictionary to hold all the keys (adjacent words) and values of the input text given by the user, so with this knowledge I came to the conclusion that this would probably require dedicated class. Next was the task of creating a new piece of text. As a result, I figured it would be a good idea to have another class dedicated to generating trigram text as well. The last required thing was a class to bring everything together. This class would be responsible for instantiating the dictionary and trigram generator objects. So I created a client class to act as a controller to create objects and call the necessary methods.

After coming up with the foundation classes necessary to produce a trigram. I figured that the client class should only have the minimum amount of knowledge to generate a trigram. In other words make calls to generate new text as generic as possible. This way if in the future someone decided to replace the dictionary implementation or text generator implementation, there would be a requirement for a huge amount of refactoring. From this knowledge, I created interfaces for the implementations to use. The class diagrams can be seen in the figure below:



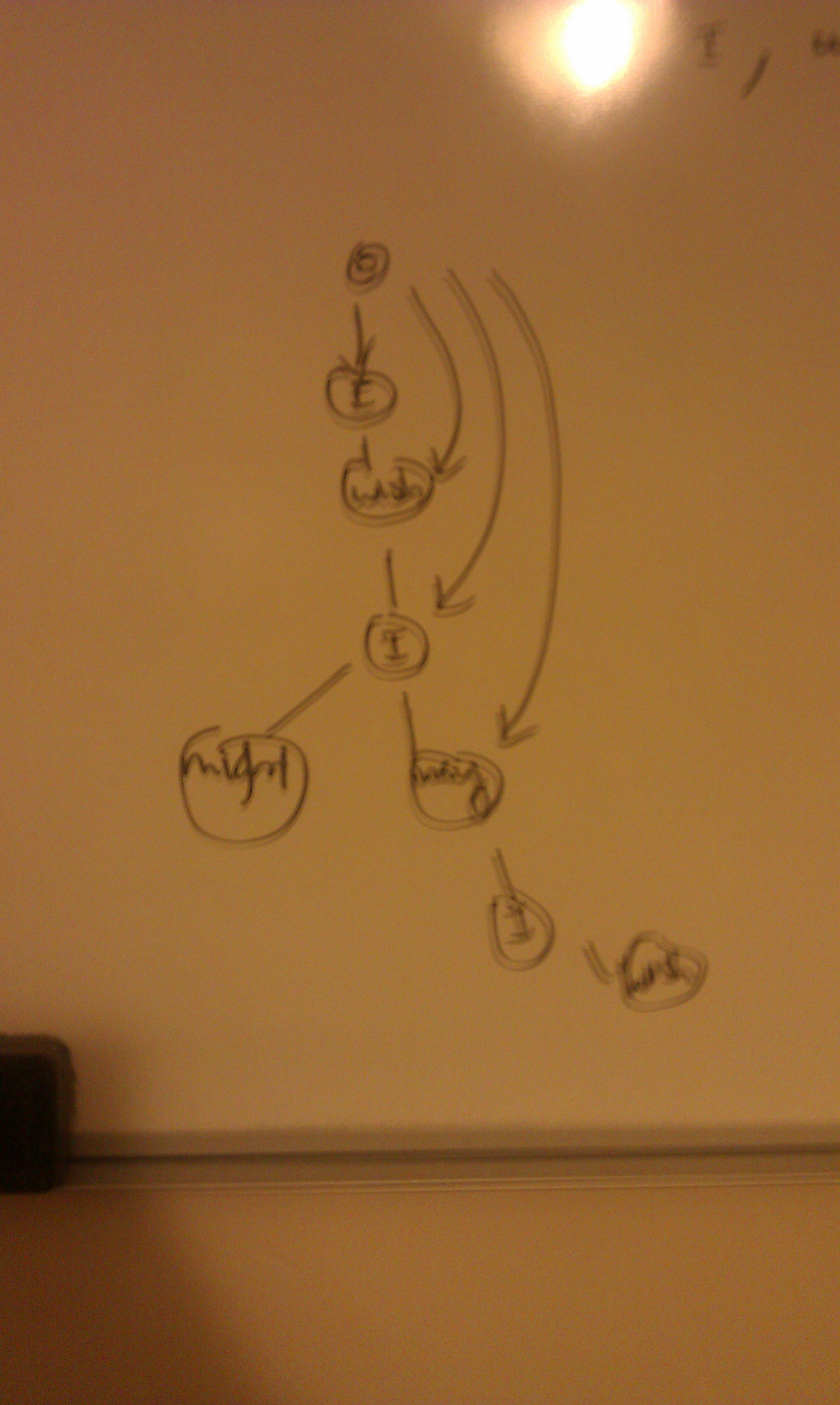
As can be seen from the class diagram. A dictionary is created with an input text file and then passed on the to the text generator to generate a new piece of text. Also, the interfaces do not have any methods specific to the trigram generation. So in essence a new implementation could be used to generate a different piece of text such as a Bigram or N-gram text.

**Dictionary Class**

**The Data Structure**

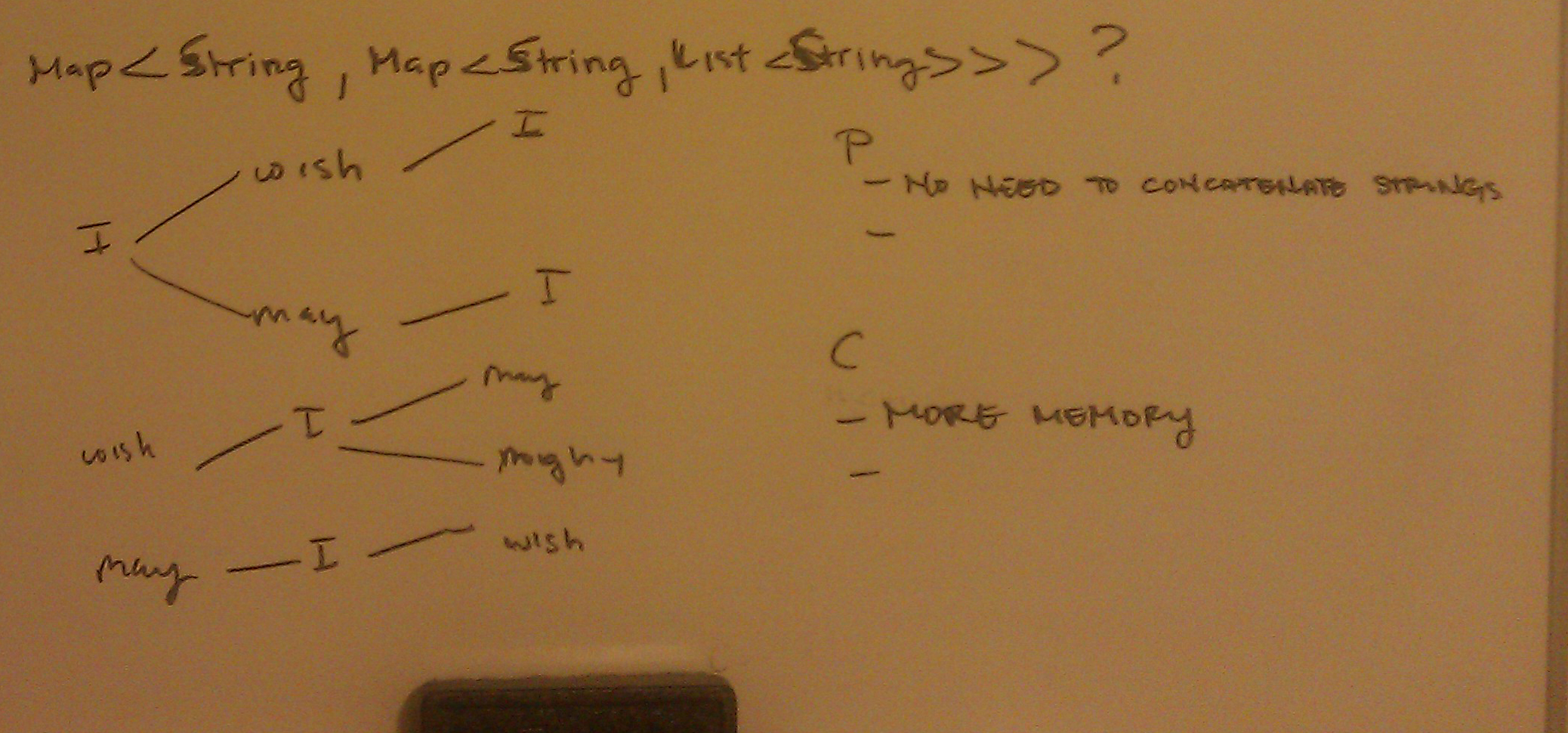
The mechanism for a trigram is to have a key and then there would be value or values associate with that key. I had two thoughts right away as for what data structure to use to hold this data. The two data structures were a hash map and a modified patricia trie.

I first went the trie route. Just from looking at the example the author of the kata give. I figure the each word could be associated in such a way that I would only have to hold each unique word once and then I could traverse the trie to find the necessary values associated with keys. I hypothesized this method could be as fast as a hash map while using a lot less memory. I tested this by drawing it on my white board as seen in the picture below:



As can bee seen from the picture, I drew a trie using the example given from the kata. The results from the modified patricia trie were that the algorithm became overly complicated (difficult to test and maintain) and the memory saving benefit became negligible, as building the trie would be loads slower. I scrapped this idea.

Next was testing the hash map. It has an O(1) search, deletion, and insertion time which is good for this type of program because there is going to be multiple searches necessary to generate the new trigram text. The drawback here was that the hash map would have to hold every single two-word combinations from the input text, so the hash map could potentially use up a lot more memory. So my next thought was maybe I could use a nested hash map that could use where the first one of the two word key would be stored in a hash map and then it’s value would be another hash map containing the second word, which then contains the final value. This design can be seen in the picture below:



Again I felt that this was overly complicated and the memory footprint would probably end up a greater deal higher than just a single hash map with a two-word string as key and a list of possible words as the value. I ended choosing “<String, List<String>” hash map. Though there can be a cause for a larger memory footprint compared to the input text I decided this would not be an issues because the input text files would most likely never be large enough to overload the heap.

**Loading Input File**

The keys to the hash map would be all the possible three words combinations in the input text. Knowing this, I design methods to take a File object and read it word by word line by line, adding all the keys and values in the dictionary.

**Trigram Generator Class**

**Starting The Text Output**

This class grabs keys from the dictionary and uses the value to generate a new piece of text. The very first key used in the output is random key. I figured it would be kind of uneventful if the output text was always the same number of combinations of words, so I created a mechanism to grab a random pair of words and then capitalize the first word so it grammar was correct. So for example, if the random pair of words to start the output was something like, “he is”, the output would start with, “He is”.

**Ending The Text Output**

There is a possible case where the very last input into the generated text does not end with some kind of sentence ending punctuation such as a period, question mark, or exclamation. For this reason, I added a mechanism to check the very last character of the input and add a random sentence ending punctuation if necessary. I also decided to keep the chances of the three punctuations selected to be equal because it made the text more interesting. I did however, come up with an algorithm which I didn’t implement to randomly grab a sentence ending punctuation according to the typical frequency of it in literature (occurrences/total words: period 7.2%, question mark 0.67%, exclamation 0.95%).

**Multiple values For A Single Key**

One key can have multiple possible values returned. I decided that it would be more interesting to have a random word selected from that list rather than grabbing the keys in order. So I decided that when a list with multiple words is returned, a random word would be grabbed from the list. This issue with this is there is a potential for an infinite loop to occur. To overcome that possible fault, I created max word output limit, which is located in a properties file that would exit the loop if it exceeded a max limit. I put it in a properties file so updates could be made without the need to recompile the code.

**Quotation Handling**

There are cases where a key with an opening quotation mark will result in several proceeding values that never close the quotation mark. To overcome this, I added a quotation flag that only enabled adding words with closing quotations if an opening quotation was added some time before that. I also added a mechanism for added the closing quotation to the very last word if it was never close.

**Paragraph Handling**

In literature the typical number of sentences is anywhere between two to ten. In the case of the book suggested by the kata (Tom Swift Under the MilkWood), the sentences were on average three sentences. From this data I gather I decided to make a random paragraph size generator that created a new paragraph after a random number of sentences that fell in a range in the properties file. So for example, if a range of three to nine is set, arbitrary paragraphs lengths between three and nine will be randomly scattered through the output text.

**Special Key Cases**

There are three special cases for keys that I accounted for:

* Keys with sentence ending punctuations (ex: “this key?”)
* Keys that have sentence ending punctuation in the middle (ex: “this. Key)
* Keys that begin with a capital letter (ex: “This key”)

To handle theses scenarios, I created a mechanism to remove the special-ness of the key and make them equivalent to their non-special key equivalent. For example, a key like, “this. Key” is equivalent to “this key” or “This key” is equivalent to “this key”. The foundation behind this mechanism is to take the value of the special key and value of the non-special key and combine them in a consolidated list where I grab a random value. After this, I post process the value to match whatever sentence criteria needed. For example, a key like, “this key?” has to have a capitalized word for its value because the key is the end of a sentence.

**App Class (Client)**

To put the text generator class and dictionary class together I decided to create a JavaFx UI. Much rather than using a regular console based client, I felt a UI was more user friend and much more dynamic in the kind of result that can be output. Furthermore, I decided to use JavaFx because it was it was a newer technology as far as UI API, standard with the release of Java 8, and I had some experience using it.

**Logging, Unit Testing, Build**

**Logging**

I believe logging to be very handy during the development process (for debugging) so I added it. I think this would also help however decides to read the code to get a gauge of how everything works. I enabled all logging levels and output them to the console just to demonstrate the logging. In normal circumstances the log would go out to a logging file and the levels would be different.

**Unit Testing**

I typically write unit tests concurrently with the class I’m trying to implements. I believe this aids with class decoupling, organization as well as enforces inversion of control. In the long run the code becomes easier to maintain. The downside is that it was a lot more time consuming than I’d like because with any test driven development, there are now two sources of code you have to update rather than one. This however, did help me a lot with debugging and making sure all the methods worked as expected.

**Maven**

I choose create the project by way of maven because it handle dependencies really well and I had a lot of experience using it.