Software Development Journal

Part of an Honors Thesis for Barrett, the Honors College at ASU

Thesis Title: Modular Tutoring Software

Author: Branden Roper

Thesis Director: Phillip Miller

Second Committee Member: Dov Zazkis

Contents

Initial Premise & Motivation

For the Application

I have always been a hands-on learner. For that reason, I have always found it kind of unfortunate that the main method with which information is conveyed in college is via lecture. Now that I am nearing the end of my undergraduate studies, I can truthfully say that, to me, a useful lecture was a rare occurrence. To be frank, I can count the number of classes where lecture was useful on one hand.

For that reason, I need to go out of my way to study to understand and retain any course-related information at all. I have researched several schools of thought on the concept of self-study, and have tried many methods of doing so. The strategy that I have had the most success with goes as follows:

(1) Obtain a piece of course material to study, usually a slideshow used in lecture.

(2) Ask myself questions about the course material as I am reading over it.

(3) Write down these questions and the answers to them.

(4) Use the questions and answers obtained in (3) to make a quiz.

(5) Quiz myself until I get all the answers right.

(6) To better retain and understand content, return to (5) at a later date for review.

In order to expedite this process, I looked at some services and software that allow the user to quiz themselves. The best one I could find ended up being Quizlet, which is a web service that allows the user to create a set of terms and definitions to quiz themselves with (think virtual flashcards). Ideally, I wanted a lightweight, open source, local application that allows the user to quickly create quizzes about any type of content with a

simple, intuitive GUI or syntax. I was unsuccessful in finding anything that met all of these criteria, so I thought to myself, “Why don’t I make it?”

For this Journal

Whenever I have a problem that I need to solve via programming, my standard process is as follows:

(1) Identify and write down the problem to solve.

(2) Check if the problem has already been solved; if so, reuse the code that solves it.

(3) Check if the problem can be solved using a function; if so, write the function to do so.

(4) Check if the problem can be solved using a class; if so, split the problem into subproblems to be solved using members of the class.

(5) Check if the problem can be solved using a collection of classes; if so, split the problem into subproblems to be solved using classes.

The actual writing of code only takes place in problem (3). When I solve this problem, I use the first solution that comes to mind. Oftentimes, this solution is not the best, simplest, or easiest solution. This solution may even have no redeeming qualities other than that it solves the problem. There may be a great deal of better solutions to same problem. Maybe the solution does not even actually work.

All of these are obviously negative possibilities, but that is not the reason I use the first solution I think of. I use this solution to better understand the problem I am trying to solve in the first place. Although it is not necessarily immediate, after writing this first solution, I almost always think of a better one. Sometimes, I do not think of a better solution for a long time (weeks, if not months). And when I implement this better solution,

I end up with less code than I started with. Often, I will implement another, even better solution after that, and end up with even less code.

One can probably tell where I am going with this by now. The more I work on my project, the less I have to show for it. So, what are some solutions to this?

(1) Keep the initial code for every problem solved because it does the job (even if it does it poorly), resulting in a larger, more impressive-looking codebase.

(2) Continue replacing solutions with better ones, and accept the fact that the codebase will grow slowly.

(3) Create some other deliverable to document the mental labor, thought process, and learning behind the project itself.

The first solution I tried was (1). Unsurprisingly, my experience went poorly. Indeed, I was able to implement the first few software functionalities surprisingly quickly. The codebase grew at a rapid pace. However, implementing each functionality became more difficult than the last, since I took no time for maintenance. Bugs were rampant, and finding their source was incredibly frustrating. Determining an ideal way for functionalities to interact was practically impossible when I did not fully understand what the functionalities were doing in the first place. Code was repeated in multiple places in multiple files, so changes to one instance of repeated code were not reflected in the others. In short, it was a mess.

For that reason, (1) was a completely unsustainable approach. I switched back to (2), which is my preferred approach, and worked on the project with great success. However, I could not shake the feeling that I would go to defend my thesis, someone would see a few files and a few hundred lines of code and go, “That’s it?” I expressed my thoughts

to my thesis director, Professor Miller, and he proposed that I create this very journal. That is how we arrived at (3), which I think to be the best approach for this project, even if it results in less time spent programming.

Minimal Requirements

At the time of submitting the prospectus for this project, I did not fully understand the workload that awaited me in the following year. For that reason, the initial requirements and timeline outlined in the prospectus were unrealistic. A more realistic, minimal set of requirements for the software will be listed here. This journal also exists as a way of supplementing the work done for this thesis.

(1) The user shall be able to write and design a quiz in a plaintext (.txt) format.

   a. The syntax for writing this quiz shall be simple enough such that it is accessible for non-programming people.
   b. The language with which quizzes can be written shall allow for the user to write questions, answers, and feedback.
      i. For a given question, the feedback section shall be optional.
   c. The language with which quizzes can be written shall allow for the user to provide randomly selected variables to be used when presenting the quiz.
      i. A randomly selected variable shall be given a name and a range of random values to choose from.

(2) The software shall be able to scan a preset directory for quiz files, and present these files to the user.

a. The software shall ignore any files in this directory that are not labeled with the extension ".txt."
b. The software shall label text files found in the quiz directory with an integer, and present this labeling to the user.
c. The software shall prompt the user to select a quiz by entering the integer used to label the quiz.
d. The software shall return to the quiz selection screen upon completion of a quiz.
e. The software shall return to the quiz selection screen if there is an error while trying to parse the quiz.
f. The software shall exit if the user enters an integer that is not a valid label for a quiz.

(3) The software shall be able to parse a quiz file, and present the quiz to the user.

a. The software shall be able to parse a quiz file, and store the data of the quiz in memory.
b. The software shall present the user with descriptive error messages if it encounters an error while parsing the quiz.
c. The software shall be able to sequentially present the user with questions of the quiz.
d. The software shall be able to sequentially accept the user's given answer for each question.
e. The software shall be able to check a given answer against the correct answer.

f. The software shall be able to provide the user with feedback regarding whether or not the user's given answer is correct or not.
    i. This feedback shall be able to be presented in two ways: immediate, and post-quiz.
g. The software shall be able to provide the user with optional feedback written by the creator of the quiz if the user's given answer is incorrect.

Java versus Python

When I initially created the prospectus for this thesis, I made the design choice of doing the entire project in Java. This, and I cannot stress this enough, was a horrible mistake. I chose Java in the first place because it was the language that I was most comfortable with. In actuality, I had not programmed anything substantial in Java since my freshman year of college. Choosing C++ likely would have been a better idea for this project, as I used that in my regular coursework in both my sophomore and junior years.

An even better idea would have been to take the risk of using this project as a means of learning Python. I will be addressing six topics in this section: code constructs, data types, functions versus methods, returning multiple values, functions and methods as arguments, and meta-programming. I will discuss the differences between how each topic is represented in both Python and Java, and how and why Python does it better every time.

Code Constructs

There are a wide variety of code constructs that are seen in most object-oriented languages. These include bodies and headers for classes, functions, methods, and loops; if, else, and else-if clauses; exception-handling blocks, and more. In Java, these constructs are organized by collections of parentheses, brackets, and semicolons. Python manages to do

most of this with whitespace, although parenthesis are still used in some places. For example, the class body consists of all lines below the body that are one level of indentation greater than the header. This makes Python much more readable, elegant, and easier to write than Java. Indented blocks are used as a conventional means of organization in other programming languages, but Python goes a step further by making it part of its syntax.

Data Types

In Java, if I want to store a value, I have to provide the type and name of the variable before I can set it to a value, and then the value has to agree with the type. Trying to set a value to a variable that does not agree with the type causes a compile-time error. This also applies to methods. I must tell Java what data type the method will return, and then I can only return a value that matches that data type. If one tries to return a data type that does not agree with the function header, they will be met with yet another compilation error.

In Python, this constraint does not exist. To create a variable, you provide a name and value; no data type is required. The variable can also be reassigned to a value of any data type. For example, a variable that holds an integer can be reassigned to hold a string, or a double, or a boolean, etc. Functions and methods operate in the same way: no return type is required, and any type of value can be returned.

Some of my fellow students criticize this feature of Python, claiming using a weakly-typed programming language can lead to error-prone code, and encourages poor programming practices. There is some weight to this criticism, as I have had moments of confusion during debugging where I thought that a variable held a value of type X, when really it was of type Y. However, variables, functions, and methods can be typed with a

feature known as “type hints,” but this feature is completely optional. In a general sense, as far as bad programming practices are concerned, bad code can be written in any language.

Functions versus Methods

The terms “function” and “method” are often used interchangeably, but by my understanding, the distinction between the two is that methods are functions bound to a class. With that being said, this is the definition that will be used here.

Java does not have functions. To write a method in Java, it must be contained within a class. The method is then bound to that class. By default, methods in Java are bound to an instance of that class (an object). However, the keyword “static” can be used, which makes a method bound to the class itself rather than a specific object. This means that if one wishes to create a function in Java, the closest they can get is creating a static method. The problem here is that there may be no logical reason for the function to be bound to a class at all. This often pollutes the codebase with boilerplate classes that only serve to hold a collection of static methods. This is also just a general nuisance that must be dealt with during the design process.

Python has both functions and methods. To create a function in Python, declare it outside of a class body. To create a method in Python, declare it within a class body. The method is then bound to that class. Additionally, Python also supports static methods via the “staticmethod” annotation This leads to less boilerplate, and allows for more freedom of design.

Returning Multiple Values

Java does not support returning multiple values. Suppose a method performs some computation based on given arguments, and this computation results in two useful values.

These values cannot be returned directly. To get around this, a Java programmer must make use of the fact that the return type of a method can be an object. So, these values must first be placed inside an object such as a list or array, and then this object is returned instead. More boilerplate and more restrictions in design; thanks Java.

Python supports both functions and methods returning multiple values. These values are given via comma-separated list.

Functions and Methods as Arguments

Sometimes, it can be useful to write a function or method that accepts another function as an argument. For this to be possible, functions and methods must be treated as first-class values. In Java, methods are not first-class. A method, as-is, cannot be passed to another method as an argument. There are two courses of action for getting around this: pass an object that has access to the necessary method, or make use of something termed a "functional interface." The first method mandates the existence of a class, while the second mandates the existence of an interface. Both result in more boilerplate unless using built-in types.

In Python, both functions and methods are first-class values. Both can be passed to another function or method by name. Methods bound to an instance of an object (non-static) passed in this way are unbound. To call such a method, one must provide an object to it via argument. No boilerplate is required for any of this.

Meta-Programming

It is difficult for me to articulate what I mean by "meta-programming." The best way I can define it is this: taking advantage of underlying code structures to an extent not immediately obvious or accessible, especially to reduce boilerplate.

For example, in Python, I was writing some test scripts. I had a collection of methods, where each method ran its own test. Then, I had a single method that would run all testing methods simultaneously. I implemented this method by manually identifying and calling each testing method by name. However, this became somewhat annoying to maintain. Every time I wrote a new test, I would have to update the method that runs all tests.

To solve this problem, I hooked into the class directory and found all names that begin with the prefix "check." These were my testing methods. Every time a testing method was found, it was called. This reduced boilerplate, and made the code a lot easier to maintain.

Surprisingly, I managed to do something similar in Java. The only difference is that I have to catch three exceptions in my run-all-tests method, and suppress a warning for every test that I write.

Closing Remarks

I could probably say even more on the topic of Java versus Python. To be honest, my complaints towards Java and praise towards Python could probably constitute an entire paper. In fact, I prefer Python so much more that in the last meeting with my thesis director before the defense, I proposed that I switch the implementation to Python. I was strongly advised against doing this for obvious reasons – changing the language of a project this late in development is a recipe for disaster. However, I genuinely believe that I may have been able to take the project even further, had I made the switch. A second, improved implementation of the software in Python could make for an interesting personal project.

Structured Learning versus Unstructured Learning

I have found that, throughout my academic career, deadlines are magical in the sense that I always manage to have at least something when they come around. That being said, by structured learning, I am referring to the mode of learning usually employed by college courses. Deliverables are expected from students at regular intervals with different modes of submission, whether it be an assignment, project, quiz, exam, etc. Deadlines are associated with these deliverables, and missing a deadline results in consequences ranging from losing a few points to failing the course.

By unstructured learning, I am referring to the mode of learning usually present when working on a personal project. There are no deliverables, no varying modes of submission, and no hard deadlines. With this thesis, there is only one deadline – the day of the defense. Even then, the author of the thesis chooses the date of the defense. So, I would say that the mode of learning for this thesis definitely falls under the category of unstructured learning.

Compared to structured learning, I find the mode of unstructured learning to be much more difficult to manage. Without regular deadlines, there was no pressure, and without pressure, motivation was not forced upon me. Does this mean that structured learning is better outright? No, I would not say that. In fact, I think that an unstructured learning experience has the potential to teach a lot more than a structured learning experience. However, it requires one to create their own structure, and one must be diligent in abiding to this structure. On future projects similar to this one, it is imperative that I create such a structure.

Lost Entries

Technically, no entries were "lost" in the sense that I created them, and then lost access to the file containing them at some later point. However, during the first semester of this project's development, I worked on the project with no version control system, and did not produce any external documentation. For that reason, within the scope of the project's development, there existed potential for me to create much more entries than can be seen here in this journal. This is truly unfortunate, because in this first semester, I was in a much poorer state of mind than I am now – which would have made for great journal material.

Retroactive Entries

Introduction

This journal was created quite late in the project's development (second to last month). For that reason, the majority of the work done on the project has already been completed. Obviously, I cannot directly capture my state of mind as it were earlier in development. However, between the first and second semester of development, I (finally) taught myself how to use GitHub, and created a repository for this project for the purpose of version control. Every update of my code has a short description to go with it. These entries contain the short descriptions themselves, their dates, and some comments from present me to give context, explanation, or my thoughts.

January 16th, 2021

Initial Commit: Just uploading what I have so far.

This one is fairly straightforward. I took the product of my work on the thesis before I started using a version control system, and uploaded it to GitHub. The classes uploaded

include Client, FileIO, Parser, Quiz, and Utility, with each class being a more primitive version of its current self.

I had used GitHub once before for a short homework assignment in an introductory software development course, but this was my first time using GitHub for an actual project. I realize it now, but at the time, I had no idea what I was missing out on. A version control system makes development a lot more approachable, and makes it a lot easier to see one's progress. Additionally, in the event that something goes horribly, horribly wrong, one can just roll back to a previous version. Frankly, I do not know why I waited this long to start using a version control system.

January 23rd, 2021

Redundancy: This file was just a wrapper for built-in Java NIO functions. I don't really know why I made it in the first place.

The file that the commit description is referring to is "FileIO.java." I ended up bringing this file back for organizational purposes. Now, it contains general purpose methods for file IO. Other classes call methods of the FileIO class to perform more specific file IO functionalities. I should have never gotten rid of it in the first place, but hindsight is 20/20.

January 24th, 2021

Redundancy: File only contained one method, which acted as a wrapper for System.out.println. I used it as a static import to make my print statements shorter, but having an entire file for that just seems kind of silly.

The file that this commit is referring to is "Utility.java." It is another file that I should not have deleted in the first place. I brought the file back with the name "Util.java," and it

contains the same print method mentioned (and more). The Util class contains general-purpose methods that I cannot logically group with any existing classes. I suppose it could be described as a class to hold miscellaneous methods.

Refactor: Cleaned up some code and wrote some comments. No major functionality changes.

This commit is pretty self-explanatory. Comments were written, code was refactored, but the software still operates in a functionally identical manner.

File Rename: Renamed to QuizParser.java.

The file that this commit is referring to is "Parser.java." Given the context of the project, one can infer that a class named Parser will parse quiz data. However, I try to avoid identifiers that prompt any sort of assumptions whatsoever. So, I renamed the file to "QuizParser.java," since the only thing it parses is quiz data.

Merge pull request #1 from bird-ron/refactor: Refactor

Unfortunately, this commit has a terrible description. The description is only one word, despite five changed files, 116 additions, and 143 deletions. I guess it was just a big refactor.

Quiz Selection: Added functionality to scan a directory and prompt the user to select one to be read as quiz data. Assumes that all files in the directory are quizzes.

This commit added a useful functionality that made quiz selection more user-friendly. Previously, to select a quiz, the user would have to type out the full name of the quiz exactly. With this functionality, the user sees which quizzes are available, and only needs to select the index of the quiz they wish to take. The functionality has been improved somewhat to only detect files with the extension ".txt."

File Rename: Renamed this file to QuizClient.java.

The file in question was called “Client.java,” and was renamed for semantics. A class named “Client” is ambiguous – what purpose does the client serve? However, “QuizClient” makes it obvious that the client is for taking quizzes. Of course, this is another place where, given context, one could infer what the file was for – but as I said previously, I try my best to avoid any ambiguity.

January 25th, 2021

Snapshot: Not sure if I’ve made any changes, but I wanted to make sure I have a snapshot of my project before I try to change things. I’m going to attempt to change the way the project parses quizzes to a token-based system, rather than just repeatedly scanning the string for tokens.

The change to a token-based system was a huge success that ended up only requiring minimal debugging. Preprocessing the data to form tokens instead of working with a massive string was also much easier for me to understand. Simply put, working with one word at a time rather than the entire data for a quiz was a lot easier to grasp.

January 28th, 2021

Token-Based Parser, Debug: Changed the parsing system from a linear-search based system (using String.indexOf) to a token-based system. Added functionality to catch erroneous tokens, but it only works for the first label in the program, otherwise it crashes. I don’t think there’s any way to handle mistyped labels (the program just interprets them as a part of the question), but I have an idea to handle misplaced labels involving regular expressions that I might implement later. Also added a debug class to store random debugging methods that I might need later.

As stated in the previous entry, the token-based parser was a success. However, just about everything else in this commit was a bust. As one can see from the commit description, the functionality to catch erroneous tokens did almost nothing. Mistyped labels (typos) ended up being a non-issue, and so the regular expression idea never amounted to anything. The debug class only contained one incredibly specific method, and at the time of writing this, it has been scrapped.

March 1st, 2021

Labeling: When parsing parts of a question (description, answer, and feedback), multiple labels can now be associated with each part. The current configuration is:

description: <description>, <desc>, <d>

answer: <answer>, <ans>, <a>

feedback: <feedback>, <fdbk>, <f>

Although technically any string can be used to label any part of a question, the use of a string not enclosed by angle brackets is discouraged.

This was another commit that added a major functionality still in use. Even the same configuration is still in use, as well. Before, the program treated labels as a single string, rather than multiple strings. The only change I would make for this functionality is something regarding the implementation. Strings that represent labels are stored as lists, which are ordered collections. However, the program could achieve the same result more efficiently using sets, which are unordered. I will likely make this change in a future commit.

Journal Entries

March 2nd, 2021

Today, I created the software development journal itself, so this will be its first genuine entry. I created the file, came up with a general structure for the document, and started typing away. I wrote the journal's first section, "Initial Premise and Motivation," which is currently comprised of two subsections: "For the Application" and "For this Journal." I think they are both quite good, and should serve as a great introduction to the journal – let us hope I think the same in a month from now!

I also wrote the section "Lost Entries," which discusses the work that was done on the project pre-GitHub. After that, I wrote the introduction to the section "Retroactive Entries." I hope to finish this section tomorrow and capture my thinking from my previous commits. Of course, commits made from this point onward will be captured in the entries in this section here.

On that note, today's thesis work consisted of a few hours of thinking and writing. The only thing I will commit today is this journal itself, although I am not sure how GitHub handles docx files. My plans as of now are to finish up this entry, give this document a quick proofreading, and then commit this document to the project's repository. Since all of today's work took place in the journal itself, this entry is definitely an example of meta-journaling.

April 6th, 2021

As one can tell from the difference in entry dates, I certainly did not "finish this section tomorrow" as was planned. This is because shortly after the last entry, my schedule was bombarded with midterms, assignments, and projects, to a magnitude that was

somewhat… unexpected. But, it has all been said before; I am here now, so let us move on, and get down to brass tacks.

At the time of writing this, the project is incomplete. It is in a workable state, but it is definitely not complete. However, it is close – with a little bit more work, the project will be in a state that I consider to be satisfactory. Although, I fear that this feeling of closeness may fall under the category of "so close, yet so far." To quantify exactly what I want to do for the software, a section will be added entitled "Minimal Requirements." I have discussed what can be accepted as a set of minimal requirements with my thesis director, and have taken some rough notes on the subject. I would like to write this section tonight.

As previously stated, this software development journal was initially made as a way of compensating for the fact that programming, and following good practices while doing so, often results in a small codebase. However, what has not been said as of yet is that, at this point, the focal point of this project has shifted away from the software itself, and towards this journal. The thesis has gone from a software development project, to a "learning about software development" project.

What does this shift-in-focus imply? Does it imply that the thesis is a failure, since development of the software did not go as far as planned? Absolutely not. A set of revised minimal requirements were defined for the software. At this time, the software is on track to meet these requirements before the defense. Even then, if by some stroke of bad luck, the thesis was not accepted outright at the time of its defense, there will still be ample time for revisions.

Another question comes to mind: suppose that software development went as planned. How much of a difference would this have made? I must admit that yes, the

software would have turned out significantly better than it is now. Looking back at the prospectus summary, the requirements were somewhat unrealistic. Would it be possible for an experienced programmer to create such an application in a year's time? Definitely. Was it possible for me to create such an application given a full course load, a capstone project, an accelerated graduate program, and additional honors work? It was not.

To summarize: although the thesis will not be up-to-par based on the original standards, it will be complete by a set of more reasonable, redefined standards that have been clearly outlined in a different section. Additionally, the focal point of the thesis has shifted to this software development journal, which was something that was not even defined in the original prospectus. At this time, I am going to break away from this entry, and write the minimal requirements section.

April 9th, 2021

Work on the project has been going well as of late. After writing the section on minimal requirements, I took the time to read over the entire document and make revisions. Only a few revisions were made, mainly for the purpose of consistency of tone. Following these revisions, I sent the list of minimal requirements to my thesis director for approval. Professor Miller gave the okay, meaning that the software now has a more concrete criteria for completion.

I have an idea for two sections I would like to add to the journal, ideally in this section of journaling. The first is "Java versus Python," and the second is "Structured Learning versus Unstructured Learning."

The first section will address my complaints about the Java programming language by comparing it with Python. Most of these complaints revolve around how certain programming tasks that are easy in Python are made unnecessarily difficult in Java.

The second section will address the difference between structured learning (such as taking a course) versus unstructured learning (such as working on a personal project). It will also address my own personal difficulties working on projects that fall into the second category.

I have added the sections that I mentioned. The first section, "Java versus Python," ended up being quite a bit longer than I thought it was going to be, and is much longer than the other. I performed minimal revisions on the first section for organizational purposes when I realized how long it was getting, but I did not perform any revisions on the other. My current plans are to finish all sections of the journal, proofread and revise, then bring the software up to par. Of course, I will continue writing entries throughout this process.

I almost forgot to mention the refactoring on the software that I did yesterday. I created new Java files to hold general purpose code for user and file IO. Other classes call methods of the UserIO and FileIO classes for more specialized purposes. I also created a Main class to hold the main method, which consists of the main menu loop. The program loops until the user selects an invalid quiz index, which registers as the command to quit. Additionally, I created a utility class Util, which holds general purpose methods with functionalities that do not align with any existing classes. The refactoring on already existing files mainly consisted of creating methods either in the same class, or in either of the added, general-purpose classes.

April 10th, 2021

This journal is proving to be a riveting success. With ease, it has grown to a length that far exceeds my initial expectations. I thought that it would be a small supplementary material to the project, consisting of a few good pages of writing. However, at this time, it has reached twenty-one pages and nearly six thousand words. I just finished close reading and revising the entire journal, and genuinely found little room for improvement.

I will be adding a simple cover page and table of contents to the journal. The table of contents will be automatically generated (thanks Microsoft Word), and will include all sections and subsections.

I am debating whether or not I want to add commit descriptions past March 1st, 2021. Commit descriptions are presented in this journal exactly as they appear in the GitHub repository between the dates of January 16th, 2021 and March 1st, 2021. However, I had planned not to include the commit descriptions past March 1st, since I would be writing journal entries anyway. But now, as I compare my journal entries to the commits made on the same day, I am finding extra bits of knowledge that might fit well in this journal.

If I do decide to add these commits, I will include them in a separate section than the journal entries themselves. Unlike the "Retroactive Entries" section, the genuine entries do not necessarily directly correspond with the commits. I fear that it would break the flow of the entire section.

The cover page and table of contents have been added. As far as the format for the cover page is concerned, I did a brief search and looked to see if there was some generalized format for a thesis cover page. However, I found that it varied from college to college, and even sometimes from program to program. So, I chose no specific format.

Rather, I just kept it simple, and presented what information I thought the cover page should present. The table of contents, on the other hand, was trivial.

I decided to add the section containing all commits past March 1st. Although somewhat redundant, I think it was a worthwhile addition. Now, my plan is to proofread and revise the sections I added to the journal since my last round of revisions (title page, table of contents, commits past March 1st, and this entry). Then, I will commit this journal, and resume work on the software itself.

April 12th, 2021

It is twelve hours before the defense. I have just now brought the software up to par based on the set of minimal requirements. From the 10th to now, I have been almost exclusively working on the software at the cost of potential journal entries. I cannot accurately describe or recall what, exactly, I did to the codebase. Additionally, I wrote thirty-three test cases and manually checked all of them. The verdict: the software appears to be working.

Does the thesis represent the application that I initially wanted it to be? No. Does the thesis represent a purely text-based proof of concept? Yes. This will likely be my final entry.

Commits Past March 1st

Introduction

Initially, I did not plan on including any commit description past March 1st. This is because there would always be a journal entry made on the same date as that commit, since the journal was created on March 2nd. However, I noticed that sometimes the commits had bits of knowledge that were only conveyed in the journal entries partially, or not at all. So, I have chosen to include the commits from March 2nd and onward here. They are in their

own section so that the information is included directly without breaking the flow of the journal entries themselves.

I was on the fence about writing this section, since it is not critically important and may contain redundancies solely because the journal entries that correspond to each commit description already exist. However, I decided that it would be better to include it for the purpose of completeness. The risk of its inclusion being unnecessary is less than the risk of its exclusion being detrimental.

The only commit not included in this section is the final commit to the repo. This is because, as soon as I commit this journal, this section is immediately outdated.

March 2nd, 2021

Add files via upload: [Software Development Journal.docx]

April 6th, 2021

Entry for 4-6-2021: Wrote a good, long entry for 4-6-2021 in the software development journal. I really need to pick up the pace!

Spellcheck: Ran spellcheck for the journal. Fixed everything that Word complained about.

Proofreading: Read over the entire journal. It's pretty good, actually. I made almost no revisions. I wrote a few notes at the end of the journal to remind myself what to do next.

April 8th, 2021

Giant Refactor: Deleted Debug.java. All it contained was a single method to print out a sequence of integers.

Created the FileIO and FileIOTests class. Other classes call methods of this class to perform file input/output operations.

Created the Main class. This class contains the main method, and allows the users to take quizzes until the quiz taking process fails.

Created the UserIO class. Other classes call methods of this class to perform input/output operations that interact with the user via console.

Created the Util class. Other classes call methods of this class to perform general, high level operations.

Other classes were modified in accordance with the newly created classes. For example, IO operations contained in the Quiz class are now contained in either the FileIO or UserIO classes.

Wrote the minimal requirements section in the software development journal. I wanted to put my software in a runnable state to make sure I did not forget anything. I have not really gotten there yet, but this commit is a step in the right direction.

Minimal Requirements: Put the minimal requirements in their own document (docx and pdf). Did this to send the requirements to professor Miller, but decided it would be good to include it in the repo as well in case I need them again.

April 9th, 2021

Java versus Python: Wrote the Java versus Python section. I briefly revised it for organizational purposes once I realized how long it was becoming, but I haven't close-read it yet.

Structured Learning versus Unstructured Learning: Wrote the section on structured versus unstructured learning, and wrote a little bit more in today's entry to go with it. Going to finish the section on retroactive entries next.

Giant Refactor Journaling: I forgot to mention the giant refactor in the entry for the 8th, so I wrote about it just now.

Java versus Python – Closing Remarks: Added the subsection "Closing Remarks" to the Java versus Python section. I basically just emphasized how much I prefer Python to Java in a general sense. I also mentioned how I proposed that I switch the implementation to Python in the last meeting before the defense. I think it makes the section much more interesting.

April 10th, 2021

Retroactive Entries, Today's Entry: I finished the section on retroactive entries, which contains all commit descriptions before March 2nd. The section is organized by date, and each commit has a secondary description to go with it. The secondary description exists to provide my thoughts and context. I also wrote a little bit for today's entry, but I may add more.

Cover Page, Table of Contents, Commits Past March 1st: Added a cover page, table of contents, and a section containing all commits past March 1st. I chose no specific format for the cover page. Rather, I just wrote what I thought should be included on it. The table of contents is automatically generated, and includes all sections and subsections. The "Commits Past March 1st" section contains only dated commit descriptions, and is separate from the Journal Entries section as to not break the flow of my writing.

Tokens, Arrays, Final: Implemented the token class and integrated it with the rest of the code. Each token consists of a lexeme, column number, and line number. This makes implementing parsing exceptions possible.

I realized that I was using array lists for everything, even when using collections of data of a fixed length. So, I switched to arrays where possible, and made a few methods that makes using them a little more convenient.

The classes Quiz, Question, and Token were full of getters and setters. They are really just used for structuring and reading data. So, I got rid of all the getters and setters in Question and Token, made their instance variables public, and made the classes immutable. I made Quiz immutable as well, but I had to keep one getter to return a question given an index.

Exceptions, Optional Feedback, Sample Quizzes: Implemented exceptions, made feedback optional, and made a few sample quizzes to test them out.

April 12th, 2021

Minimal Requirements Fulfilled, Test Cases: In a frantic programming frenzy, I managed to fulfill all minimal requirements for the thesis. I also wrote a few test cases. This description would be better, but I forgot most of what I changed.