Overcoming Code Rot in Legacy Software Projects

by

Austin Jun-Yian Liew

S.B., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

© Massachusetts Institute of Technology 2017. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 26, 2017

Certified by . . . .
Una-May O'Reilly
Principal Research Scientist
Thesis Supervisor

Certified by . . . .
Erik Hemberg
Research Scientist
Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Overcoming Code Rot in Legacy Software Projects

by

Austin Jun-Yian Liew

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Legacy software projects tend to be the most susceptible to code rot due to typical issues (e.g., outdated documentation) that affect software projects over the course of their lifetimes. While software engineering has matured over time to promote good practices (e.g, Test Driven Development) that prevent such issues, oftentimes developers will still have to inherit legacy code that is in the late stages of code rot. It is not always feasible to replace the inherited legacy code with a new project developed from scratch. Furthermore, inheriting legacy code often brings along additional new requirements that must be fulfilled.

The translation software under the MOOCdb project is a representative case of the aforementioned scenario. The MOOCdb translation software aims to unify course data from different MOOC (Massive Open Online Course) providers into a single format named MOOCdb. However, the translation software lacked stability and up-to-date documentation. The author of this thesis was tasked not only with getting this software operational, but also implementing new functionality to respond to ongoing demands of collaborators from another institution.

This thesis presents the application of the LCCA (Legacy Code Change Algorithm) on the legacy MOOCdb translation software project to solve its code rot and make it maintainable. The evaluation is based on the rate at which issues were resolved during different stages of LCCA, and an analysis of the key differences after the application of the LCCA. We see that the project saw much more frequent changes in response to discovered issues once the project was significantly refactored and restructured, and that the stability is greatly increased with the addition of a test harness to the core modules. Such changes enabled industry accepted good practices and alleviate the bulk of the code rot that prevented the project from moving forward.

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist

Thesis Supervisor: Erik Hemberg
Title: Research Scientist

// Acknowledgments

I would like to thank Erik Hemberg for his supervision of my research work. He allowed me great flexibility in how to do my work and also flexibility in scheduling our meetings, many of which we initially spent trying to understand what the MOOCdb project actually did in practice versus what it did in public-facing descriptions. I think we both gained a valuable amount of knowledge and experience working on this project together.

I would also like to thank Una-May O'Reilly for granting me the opportunity to work in the ALFA group. I am not sure what I would have done without the opportunity but I have no doubt that it would have resulted in a much less successful project. She also allowed me to return to a fragment of my youth in Hong Kong, which is proceedingly less likely for me to return to as the years pass by.

I would especially like to thank my mother and father who granted me many of my childish desires growing up, and allowing me to proceed through all of my education debt-free. In retrospect I wish I was a better behaved child growing up, so I hope I can make up for it proceeding onto adulthood as a more responsible member of society.

I would also like to give a shout out to my brother Vincent, whom I have always looked up to as an example of academic excellence. For my academic dreams I left behind, he's managed to live them out and the sight of this leaves me in a state of awe every time I reflect on it.

Last of all, I would like to thank my friend Anton from Belgium. Sometimes when I do not feel too great about my endeavors in learning Japanese, I remember that at least I have managed to influence others who are pursuing the same endeavors into aiming for their best and in some ways, that alone has made the struggle worth it. Anton is a model example of such an individual I have influenced in this way. I hope he achieves his dream of naturalizing as a French citizen, and I enjoy every picture of his biceps or curry he sends me over IRC.

Contents

1 Introduction **13**

1.1 Motivation . . . 13

1.1.1 Clean Code . . . 13

1.1.2 Code Rot . . . 17

1.2 Research Question . . . 20

1.2.1 Legacy Code Change Algorithm . . . 20

1.2.2 Design Goals for MOOCdb `translation software v1.4` . . 21

1.3 Contribution . . . 22

2 Background **25**

2.1 Evaluating Code Cleanliness . . . 25

2.1.1 Understandability . . . 25

2.1.2 Resistance to Change . . . 26

2.1.3 Stability . . . 26

2.1.4 Reproducibility . . . 26

2.2 MOOCdb and `translation software 1.0` . . . 27

2.2.1 apipe . . . 29

2.2.2 qpipe . . . 29

2.2.3 curation . . . 30

2.3 Collaboration and Demands . . . 30

3 Evolution of `translation software` **33**

3.1 Bug Fixing . . . 33

3.1.1 apipe 34
3.1.2 qpipe 34
3.1.3 curation 35
3.2 Refactoring 36
3.2.1 Code Reorganization 37
3.2.2 Code Reformatting 39
3.3 New Functionality 40
3.3.1 Internally Motivated 40
3.3.2 Externally Requested 41
3.4 Test Harness 45
3.5 Documentation 47

4 Evaluation of `translation software v1.4` 49
4.1 Understandability 49
4.2 Resistance To Change 52
4.3 Stability 53
4.4 Reproducibility 57

5 Conclusion 61

6 Future Work 63
6.1 Continuous Integration 63
6.2 Automatically Generated Documentation 64

List of Figures

1-1 Good Code Organization 14
1-2 Bad Code Organization 15
1-3 Inconsistent Code Indentation 16
1-4 Consistent Code Indentation 16
1-5 Test Driven Development Flowchart [3] 18
1-6 Cost of change over project lifetime [4] 19
1-7 Legacy Code Change Algorithm [7] 21

2-1 MOOCdb Data Transformation Flowchart 27
2-2 `full pipe v1.0` Flowchart 28

3-1 Code Organization for `translation software v1.0` 37
3-2 Code Organization for `translation software v1.4` 38
3-3 Portion of `full pipe` Control Flow Graph 42
3-4 Core MOOCdb `v1.4` and `vismooc_extensions` tables 43
3-5 `moocdb_index` tables 44
3-6 `moocdb_index` query for a user in more than one course 44
3-7 `full pipe v1.4` Flowchart 46

4-1 `pylint` scores for `qpipe` `v1.0` and `v1.4.4` 51
4-2 `pylint` scores for `curation` `v1.0` and `v1.4.4` 52
4-3 Number of Commits v. Date 53
4-4 Scatterplot of Coverage vs. Statements for `qpipe v1.4.4` 58
4-5 Scatterplot of Coverage vs. Missing for `qpipe v1.4.4` 58

4-6 Scatterplot of Coverage vs. Statements for `vismooc_extensions v1.4.4` 59

4-7 Scatterplot of Coverage vs. Missing for `vismooc_extensions v1.4.4` 59

List of Tables

3.1 Bug Summary 33

4.1 pylint Score Comparison for translation software 51

4.2 qpipe and vismooc_extensions Test Count 54

4.3 qpipe v1.4.4 Test Coverage 56

4.4 vismooc_extensions v1.4.4 Test Coverage 57

Chapter 1

Introduction

In this chapter we discuss good software practices and the negative consequences of failing to adhere to such practices. The proposed solution in this thesis for resolving such consequences is also explained. Lastly, we also summarize our contributions to this research topic.

1.1 Motivation

1.1.1 Clean Code

The exact definition of clean code varies between individuals but it is generally agreed that clean code is code which is well organized, well formatted, and performs its purported task well and accurately according to specifications described in documentation [1].

Code Organization

Code organization refers to the arrangement of code across several files, and where those files are placed in a project folder. While the actual placement is arbitrary in the context of the actual execution of the code, the quality of code organization can greatly affect the difficulty of maintaining the project's codebase. The quality also affects the difficulty of bringing collaborators into the project. A tree-like structure is

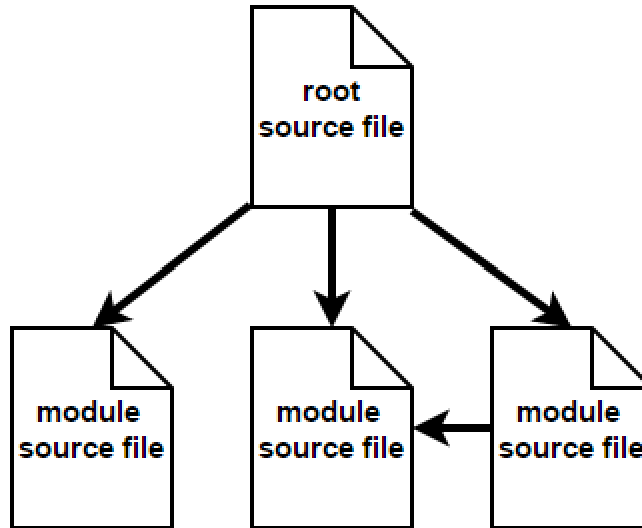

Figure 1-1: Good Code Organization

typically recommended where modules in source files only import code from other files in the same directory or in subdirectories relative to the module under observation by a developer. This allows an individual trying to understand the project's code to begin at the starting point (the topmost file in the tree) and only go downwards as more details are necessary. On the other hand, if code further down the tree ends up importing code from higher up the tree, the code flow becomes obfuscated. Such an organization would mean that the level of code abstraction can no longer be approximated by how high or low the source file is on the tree. Hence is it desirable to stick to a tree structure as much as possible. Figures 1-1 and 1-2 illustrate the two scenarios described.

Code Formatting

Relatedly, the formatting of the lines of code also affects the difficulty involved in understanding the code. While there is no single universally agreed upon format which is the clearest, it is more important to maintain consistency of a reasonable format. Figure 1-3 illustrates an inconsistent indentation between two different conditional blocks whereas Figure 1-4 illustrates the same scenario but with consistent indentation. Maintaining a consistent format throughout the code makes it easier

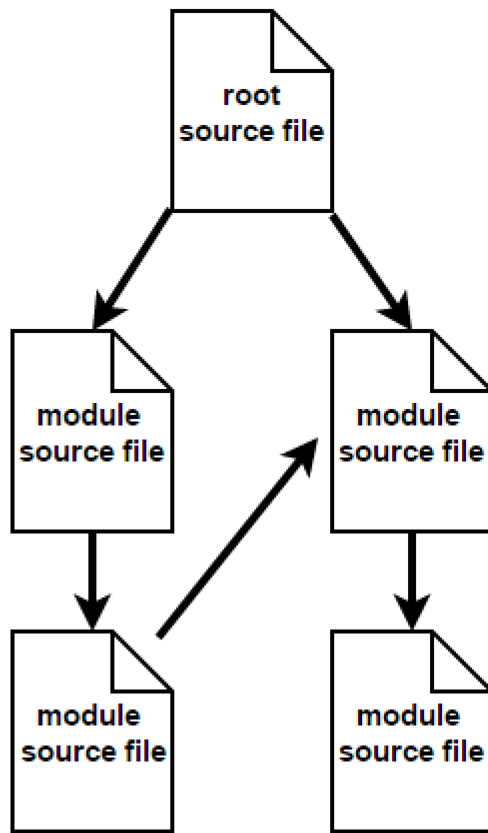

Figure 1-2: Bad Code Organization

```
a = 0;
b = 1;
if (a == 0) {
    a = 1;
b = 2;}

if (a == 1)
{
b = 3;
a = 0;
}
```

Figure 1-3: Inconsistent Code Indentation

```
a = 0;
b = 1;
if (a == 0) {
    a = 1;
    b = 2;
}

if (a == 1) {
    a = 0;
    b = 3;
}
```

Figure 1-4: Consistent Code Indentation

for collaborators because they can get accustomed to a single layout and focus on understanding the actual algorithms represented by the code.

Testing and Documentation

Naturally it is important that code performs the task that is intended. However, without proper documentation it is not always clear what is intended. One cannot determine if a piece of code is correct or incorrect without knowing how correctness is defined in the context of the project. Hence it is important that there are both human-readable comments to explain the purpose of code modules and also ex-

plicit tests that define and guarantee correctness under a variety of scenarios. The human-readable comments help collabators understand the high level ideas of the code modules while explicit tests demonstrate the intended usage of different code modules. TDD (Test Driven Development) is often recommended as a practice that regularly enables efficient development while maintaining a sturdy set of tests that is consistent with documentation. Figure 1-5 illustrates a flowchart of TDD. The key point is that the developer does not write any implementation until a failing unit test has been written. Once a unit test fails, then the bare minimum code necessary is written to pass that and all previous unit tests. This cycle continues until the desired functionality has been obtained. Following this practice guarantees that no code is left untested, guaranteeing that correctness is maintained as the project is extended.

The utility of up-to-date unit tests and documentation also extends to collaborators. In a brief survey of MOOC related software, the author tried to run the moocRP [2] software but was not able to even start the code. This was a consequence of install instructions being out of date and the software (largely javascript) libraries having progressed much further than the original versions used. While academic software is known to omit proper testing and documentation, these elements should be prioritized to maintain academic integrity. It is important that results are reproducible in academia and when software is involved, reproducibilty is pinned to whether or not the software can be run at all.

1.1.2 Code Rot

There are many kinds of good software practices that can vary depending on the end goal but they are largely informal. Examples include consistent naming conventions, maintaining portability of code across different environments, and comprehensive unit testing. If good software practices are not upheld throughout the development of a project, then a phenomenon known as code rot occurs.

Code rot is the phenomenon of poorly designed and implemented code leading to the growth of bugs and increase difficulty of maintenance [1]. Poor design and poor implementation can take many forms but generally it is code that violates all

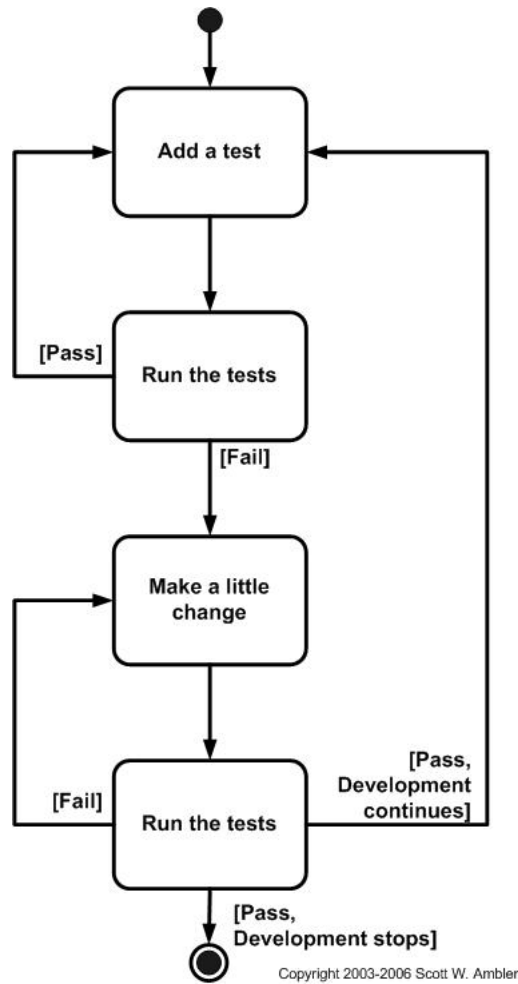

Figure 1-5: Test Driven Development Flowchart [3]

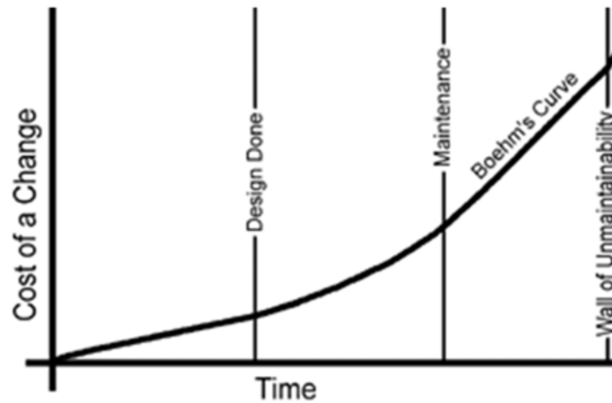

Figure 1-6: Cost of change over project lifetime [4]

the aforementioned properties of clean code. For example, a method with a very strict interface that is used throughout an entire software project would typically be considered bad design because it is resistant to change due to its strict interface being forced throughout the project. From poor design, often when one bug is fixed, subtly several more bugs are introduced into the code. This raises the maintenance cost significantly such that fixing one bug or trying to extend the functionality of the project while maintaining correctnes incurs large overhead. The lifetime of the project may see the smooth initial development slowly degrade until it becomes so difficult to modify that the project is abandoned, signaling death from rot. Figure 1-6 charts this phenomenon.

Robert C. Martin's book on clean code [1], widely considered to be the authoritative source on the subject, was originally published in 2008. Yet, people are still writing about the subject as recent as 2017 [5]. Much online discussion [6] can be found regarding the still fairly wide spread of rotting code since Martin's book on the subject and the opinion largely supports precisely the same principles Martin laid out in his book. There is widespread agreement that a lot of rotting code is due to developers or scientists cutting corners to meet deadlines quickly. Putting TDD into practice also requires careful forward thinking about code design. It is simpler for a developer to solve exactly the problem he or she has in mind but to anticipate potential roadblocks or changes in requirements is not nearly as straightforward. Hence, despite the agreement that the principles of clean code are very important, the reality

is that a lot of rotting code still exists and gets used.

1.2 Research Question

In an ideal world everyone would follow good software practices and code rot would be non-existent throughout all software projects but the reality is that maintenance of good practices is frequently sacrificed to the demands of getting new functionality implemented quickly. Hence, oftentimes the reality is that software engineers want to either modify legacy code to fix broken functionality or provide new functionality; at the same time clearly the ability to develop using a TDD approach is desirable as it provides confidence that an existing implementation is correct and provides ease of extendibility.

The general research question of this thesis is "What is the best way to continue the development of legacy code that rots?". A solution to this problem must both be able to undo the damage from code rot while being able to respond to new demands for the legacy code. We applied an algorithm known as the Legacy Code Change Algorithm on a software project developed in our group known as `translation software` for the MOOCdb project. Given the `translation software` project, we want to determine if the application of LCCA will yield demonstrable gains in preventing code rot and obtaining quality clean code.

1.2.1 Legacy Code Change Algorithm

The LCCA (Legacy Code Change Algorithm) [7] aims to solve the aforementioned scenario. This algorithm captures the end goal of making meaningful changes to the legacy code base while also improving it over time. The hope is that with this method, a large monolithic legacy project can slowly be broken down into smaller modules with testing contained in the areas with code changes, and eventually reach a state where further development can be performed using the TDD methodology. The algorithm is summarized in Figure 1-7. Eventually steps 4 and 5 should be done in reptition and alternation to mirror the TDD methodology. In step 1, change points

1. Identify change points
2. Find test points
3. Break dependencies
4. Write tests
5. Make changes and refactor

Figure 1-7: Legacy Code Change Algorithm [7]

refer to the locations in code which must be changed in order to either provide a new feature or to eliminate a bug. In step 2, test points are the locations in code where code can be added to provide a test for the changed code. In step 3, dependencies usually take the form of large monolithic sections of code that could be separated and rewritten to split up the dependencies into independent modules.

LCCA was selected as a means to combat code rot because much of the general advice about how to handle legacy projects with code rot either provide miscellaneous tidbits of advice, or refer to the LCCA as the recommended method. The LCCA was an appropriate choice for this thesis study because it outlined specific steps that could be applied systematically in order to remove code rot. In order to evaluate the effectiveness of LCCA we had to define a measure of code rot. We decided to evaluate its success from the perspectives of understandability, resistance to change, stability, and reproducibility (see Section 2.1).

1.2.2 Design Goals for MOOCdb `translation software v1.4`

When setting out to apply LCCA to MOOCdb's `translation software`, we wanted to work towards concrete design goals for the target `v1.4` release at the end of this thesis study. While the ultimate end goal of the whole MOOCdb project, including `translation software`, is to provide a unified format for MOOC course data that can be used easily by data scientists or researchers with less programming experience than a fully fledged software developer, the goal for the end of this thesis study was to ensure that `translation software` was left in a state that was not only **usable**

but could also be more readily **extended** and **maintained** by a developer with less experience and weaker programming skills. Specifically, `translation software` is scheduled to be inherited by a rising third year undergraduate student over the summer semester. The thesis author on the other hand had already completed his 4 year undegraduate program and acquired significant and diverse software skills during summer internships. Since it cannot be assumed that the successor for the project knows all the tricks of the trade, it was imperative to reduce as many obstacles as possible that might get in the way of a developer trying to familiarize him or herself with the project on the source code level. Simply put, it was necessary to eliminate as much of the code rot as possible during the year long thesis study.

1.3 Contribution

The study undertaken for this thesis provides some insight into the challenges of adopting LCCA on an actual legacy software project. This thesis documents the challenges encountered and the solutions applied to said challenges. The challenges are summarized as follows:

- How can one uncover and clarify obfuscated code flow?
- What is the most efficient method of unifying an inconsistent code style?
- How should one reorganize a code base whose dependencies between modules are unclear?
- How does one address demands for new functionality when trying to eliminate code rot from the existing functionality?

The hope is that this serves both as a useful study for developers in general and as a useful document for future maintainers of the software project upon which LCCA was performed.

We find that the `translation software` improved significantly from its `v1.0` state to its `v1.4.4` state in terms of understandability, resistance to change, stability,

and reproducibility. The definition of these terms can be found in Section 2.1. A large part of the improvement was due to useful automation tools that helped eliminate inconsistencies in the source code as well as enable a test harness to maintain the intended functionality of the software. We also find that using TDD as much as possible allowed for higher test coverage, which improves stability more than LCCA will necessarily provide.

Chapter 2

Background

In this chapter we discuss our LCCA evaluation choices, the initial state of the `translation software`, under the MOOCdb project, the high level function of the different components of the translation project, and also the nature of the collaboration with which the author was engaged during the maintenance of the `translation software`.

2.1 Evaluating Code Cleanliness

In this section the criteria we use for evaluating code cleanliness is summarized. There are other criteria typically included in the evaluation of code cleanliness such as portability; however, we chose to focus on these four criteria as they covered the core issues we wanted to address under the time constraints of the thesis.

2.1.1 Understandability

Understandability is a measure of how easy source code is to understand. We have chosen to separate this into two categories: code flow and visual clarity. Understandability with regards to code flow addresses how easy it is for a person to determine which source files of the project are being executed during the execution of software. Understandability with regards to visual clarity addresses how easy it is for a person

to look at a single source file and quickly grasp the algorithm being executed in a small section (e.g. a helper method). We use `pylint` scores to quantify the readability (i.e. visual clarity) of the `translation software`.

2.1.2 Resistance to Change

Resistance to change is a measure of how difficult it is to modify the existing source code. A major negative property of rotting code is that it is resistant to change. Large monolithic methods are common examples of this kind of resistance, as oftentimes the dependencies between different sections within the method are not explicit. We use the commit activity over the duration of this thesis study to support our claims that the recent versions of the `translation software` have reduced resistance to change. We also also describe a collaboration experience when developing new functionality to support this claim.

2.1.3 Stability

Stability is a measure of how well supported software is by a test harness so that bugs that shift the software away from its intended functionality can be quickly identified and eliminated before it propagates too far down later revisions. We use unit test counts and test coverage as a means of quantifying stability.

2.1.4 Reproducibility

Reproducibility is the capacity for the code to consistently reproduce results given fixed inputs. To this end, clearly defined versions of the software are also necessary as it provides an exact state of the code upon which one can recreate outputs given the same set of inputs someone else used. We use collaborative experiences while developing the `translation software` to support our claims that the recent versions of the `translation software` enable reproducibility.

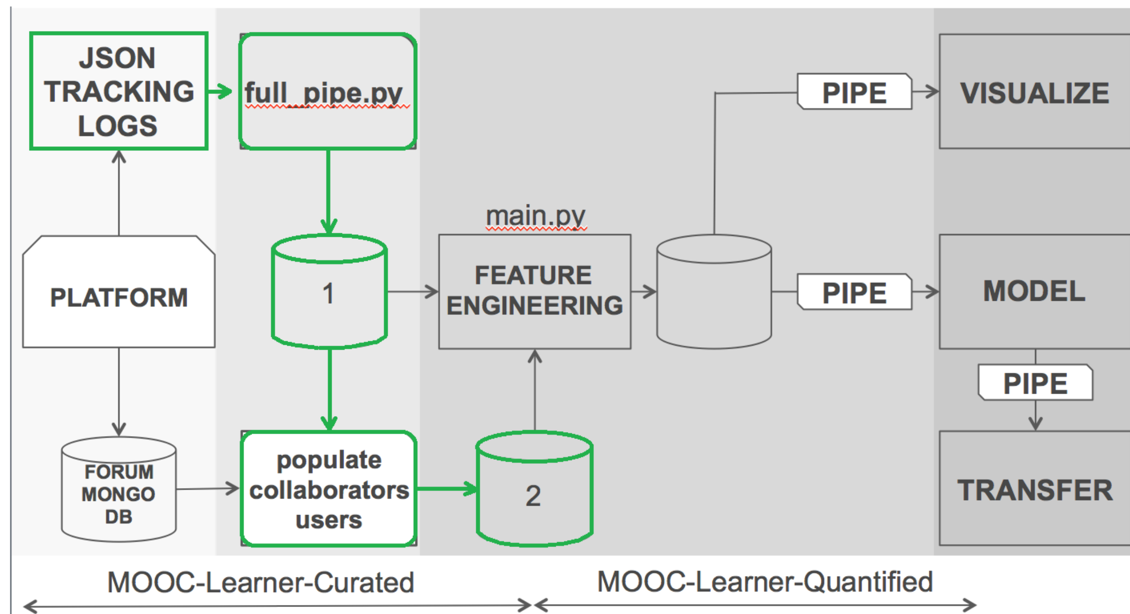

Figure 2-1: MOOCdb Data Transformation Flowchart

2.2 MOOCdb and `translation software 1.0`

The purpose of the MOOCdb project is to provide a unified format for MOOC course data which would enable education researchers, computer science researchers, machine learning researchers, technologists, database and big data experts to perform studies and analysis on such data more easily [8]. Figure 2-1 flowcharts the process of taking course data from a raw tracking log to a database format which can then be used for feature engineering. The path highlighted in green (starting from the `JSON TRACKING LOGS` block and ending at the 2 cylinder) is internally called the `translation software`.

The `translation software` under the MOOCdb project is a crucial component. The translation project takes MOOC specific course data (specifically the tracking log) and converts it to the MOOCdb schema and stores it in a MySQL database. The primary support is for the Edx course data; during some previous iterations of the software, Coursera was also supported. Since MOOC specific course data tends to include a lot of information specific to that particular MOOC platform, the ultimate goal of the `translation software` is to unify different MOOC course platforms' data into a schema that is general enough to capture essential data that is common across

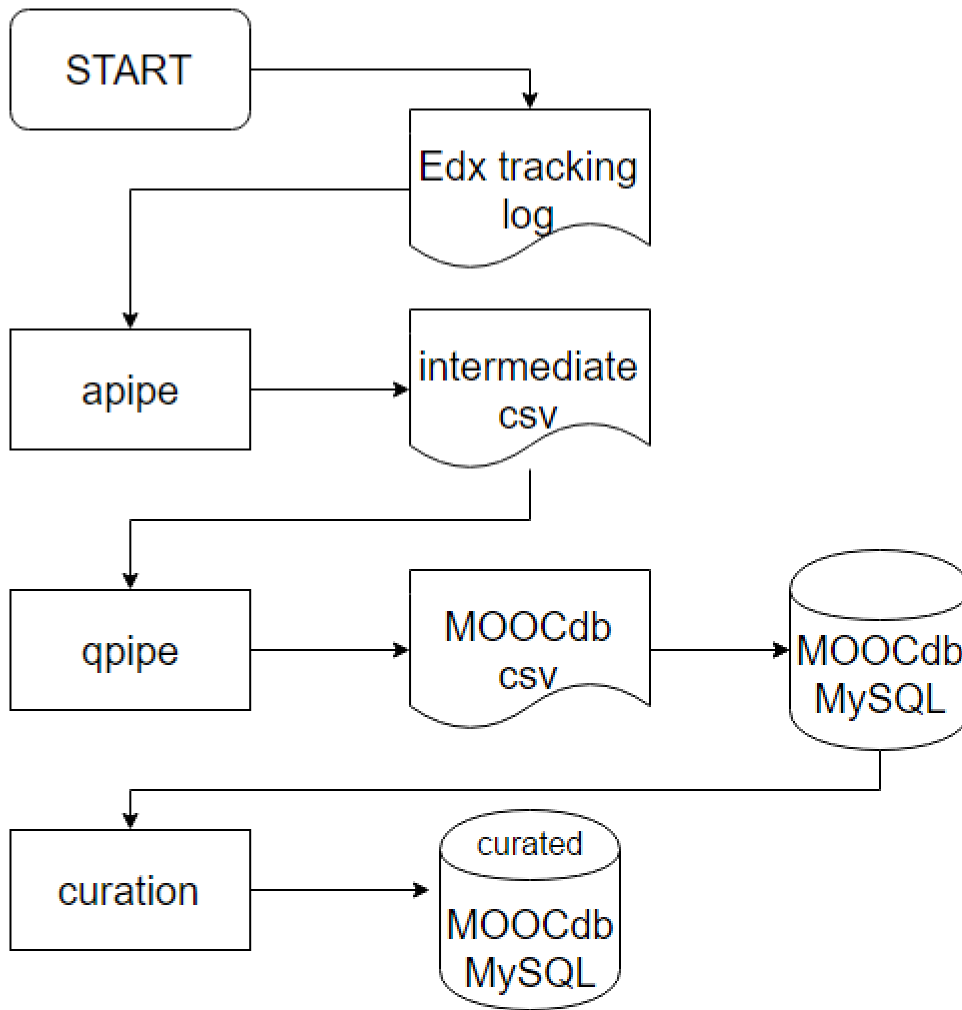

Figure 2-2: `full pipe v1.0` Flowchart

different hosts.

The entry point to the `translation software` is internally called `full pipe`. `full pipe` is just a user-facing script that ties together the core components of the `translation software` together and eventually outputting a MOOCdb MySQL database. Figure 2-2 illustrates the data flow across `full pipe`. Code components are aligned on the left, while files or db output are aligned on the right. The different components are described in the following sections.

A big emphasis of the MOOCdb project is to eliminate personally identifying information from the course data so that data could be shared between different

institutions. This is necessary due to strict rules regarding privacy violations in academic data used for research. Hence, the `translation software` also had to do the work of anonymizing user data. This job is delegated to `apipe`.

Since previous revisions of the `translation software` had no notion of explicit versioning, we define `v1.0` to be the last meaningful commit[1] before this project was inherited by the thesis author.

2.2.1 apipe

`apipe` is a Python library used to convert a raw Edx tracking log file to an intermediate format from which it is easier to parse and extract data. `apipe` is the name used within the `translation software` but it is technically an open source library called `json_to_relation` developed by Andreas Paepcke [9]. An edx tracking log file is fed to `apipe` which then outputs an intermediary csv format. This intermediary csv format provides a simpler tabular format that is easier to manipulate later. It is also not uncommon for Edx tracking log entries to be malformed; in this case those entries cannot be used at all. Hence it is easier to perform the transformation to the MOOCdb schema from the cleaner, intermediate format. Besides this function, `apipe` also cleans a lot of the Edx tracking log data to anonymize usernames via hashing and converting IP addresses to country codes. This satisfies the necessary aforementioned privacy requirements on sharing academic data.

2.2.2 qpipe

`qpipe` is a component of the translation project originally developed by Quentin Agren [10]. It takes the intermediate format from `apipe` and performs the necessary transformations to convert it to the final MOOCdb schema, and outputs csv files which are then imported to a MySQL server. Some transformations are simpler than others. For instance, taking the anonymized username from the intermediate Edx tracking event table from `apipe` is a direct map to a `user_id` in the MOOCdb schema.

[1] commit hash `043a10217d6b9dc6eb9f2aaace6eb41ebaf4d8d2`

However, problem ids in in the intermediate Edx tracking event table are altered from their original id and so they must be reconstructed. For example, `apipe` might generate a problem id `i4x-MITx-6_002x-problem-H10P2_New_Impedances_10_1` which must be restored to `i4x://MITx/6.002x/problem/H10P2_New_Impedances/10/1/` by `qpipe` to match the Edx URI properly.

2.2.3 curation

The `curation` component of the translation project is responsible for cleaning up the data ouput from `qpipe` to better enable data science analysis. It operates on a MOOCdb MySQL instance and specifically cleans up the observed events and submissions tables, flagging entries as either valid or invalid based on timestamp data associated with the entries. For example, it may not be desirable to have consecutive repeated events in the MOOCdb observed events table. Consecutive repeated events are defined as consecutive events where the anonymized user id that triggered the event, the timestamp of the event, and the duration of the event are all the same across consecutive events. It may not be desirable to have such duplication of events as it may create unnecessary noise that could affect feature engineering performed later. Consecutive events after the first such event are marked as invalid so they can be filtered out in a MySQL query.

2.3 Collaboration and Demands

This research was being conducted in parallel with collaborators from other institutions. We were actively collaborating with HKUST's VisMOOC [11] data science team and the GPU team, but primarily the former. During the maintenance of the `translation software`, it was necessary to ensure that the VisMOOC and GPU teams were able to run the software on their end and that it worked with their Edx course data. Furthermore, we had to respond to demands from the VisMOOC team for new functionality that they required to interface with their visualization software. Section 3.3.2 elaborates on the nature of the new functionality required. We view

extensions as the solution for collaborative demands as it allows us make a clean separation between the core MOOCdb `translation software` logic while providing a simple interface for other groups to implement their own flavors of MOOCdb for their purposes. Section 3.3.2 elaborates on the extensions we implemented for the VisMOOC team.

Chapter 3

Evolution of translation software

This chapter details the evolution of the translation software throughout this thesis study. Sections 3.1 to 3.2 comprise the execution of steps 1-3 of LCCA (Figure 1-7): identify change points, find test points, and break dependencies. Sections 3.3 to 3.5 comprise the execution of steps 4-5 of LCCA: write tests, make changes and refactor.

3.1 Bug Fixing

This section describes the kinds of bugs that were fixed in the translation software during this thesis study. apipe had some, but not too many bugs and could be used as a library viewed as a black box. qpipe had by far the most bugs which were the most important to resolve because the bulk of the translation software specific logic occurs in qpipe. Table 3.1 lists the change points (module) for each of the described bugs in the following section.

Pipe	Bug Name	Change Point
apipe	Unescaped Single Quotations	json_to_relation/edxTrackLogParser.py
apipe	Stringified NoneType	json_to_relation/edxTrackLogParser.py
qpipe	User-input Source Code	edx_pipe/edx_pipe.py
qpipe	Single Timestamp Format	qpipe/genformatting.py
qpipe	Unspecified Esacpe Character	qpipe/extractor.py
curation	MySQL Lock Timeout Exceeded	curation/submissions_curation.py

Table 3.1: Bug Summary

3.1.1 apipe

The modifications made to `apipe` are internal to the `translation software` and do not affect the public-facing project maintained by Andreas Paepcke [9].

One of the major functions of `apipe` is to convert the raw Edx tracking log to a cleaner intermediate format. It is common for a noticeable number of entries in the raw Edx tracking log to be unuseable due to the raw data being malformed, or not escaping special characters correctly (`MITx 2.03x 2013 Spring` for example has 350,000 unusable entries of an Edx tracking log containing 6,000,000 entries). Hence, `apipe` attempts to make a best effort parse and conversion of the entries in the raw Edx tracking log. However, it is not perfect so some minor modifications to the event type case handler were made to improve the parser in `apipe` so that it would be able to handle previously unhandled cases. Some case handlers were changed because they would not escape characters such as single quotations properly which would further cause problems in `qpipe` when trying to read the intermediate csv data.

A different bug would create false positives in `qpipe`. Some fields in the intermediate csv would be populated with the string "None" when it really should have been an empty string or a NoneType value. This is problematic because `qpipe` will check this value to decide whether to perform further processing on it. An empty string or a NoneType value would be interpreted as False boolean value in Python, whereas the string "None" would be interpreted as a True boolean value. As a result, `qpipe` would treat the value as if it was valid when it is not.

3.1.2 qpipe

The following is a selection of the most notable `qpipe` bugs.

One of the early bugs fixed was one where a user would be prompted for login credentials for the MySQL instance on the host machine. The way this was implemented was using the `raw_input` built-in method in Python; this evaluates the input exactly the same as if it were source code. This is very non-standard because it requires a user to know Python programming details instead of simply being able to enter in a user-

name and password. This behavior was also inconsistent with other user-interfaces in the project and so it was changed to evaluate all user input as strings.

Another bug occurred when timestamp data in the intermediary csv files would not match a timestamp format configured in the source code. This was a bug that only affected Edx `6.002x 2012 Fall` course data, where a very small number of entries would have an alternate timestamp format. The reason for this is unclear, but the fix was simply to attempt using several timestamp formats until one matched. This fix was chosen as at most one timestamp format would be able to match a timestamp value.

The trickiest bug to resolve was one whose effects emerged in two separate issues. One issue started with modifying the code to accept a possible value for an Edx assignment submission entry that is not documented on the Edx developer-facing documentation. While this was not a difficult fix, afterwards many errors appeared as a result of `nan` values being found. `nan` is a contraction of "not a number" which is specific to float data types; this added to the confusion because the data should only be string data. At the same time it was found that the lookup key used to find particular submissions would not match any entries in the intermediate csv file containing all the submissions. It turns out that both of these issues were a consequence of one of the Python libraries (`pandas`) being used incorrectly. The escape character
had to be specified explicitly in order for `pandas` to read the intermediate csv files correctly. Without this, once a submission entry was found with a
character, the fields read into `qpipe` would no longer be in order and so practically all entries afterwards would not be mapped to the correct field. This bug caused a large bulk of the data read into `qpipe` to be nonsensical. This bug demonstrates quite well that even bugs that have small fixes can cause a non-trivial amount of damage.

3.1.3 curation

The `curation` bugs were lower priority than `apipe` or `qpipe` bugs because the effects of `curation` are mostly optional as far as the MOOCdb schema is concerned. Nevertheless we discuss one of the bugs since it illustrates that an ideal solution cannot

always be obtained.

When the course data is too large then the `curation` code fails because a built-in MySQL lock wait timeout has been exceeded. Initially we considered simply increasing the length of the lock timeout so that it would work. However, this would then require intervention on the part of any users of the `translation software` to configure their MySQL software from the defaults. Furthermore, they may not be using MySQL solely for MOOCdb so in the interest of minimizing footprint of the `translation software`, we decided not to pursue this option in the end. Then we considered breaking up the problematic SQL database transaction that the `curation` code performs into smaller chunks so as not to exceed the timeout. Ideally we would like to split the transaction into equally sized chunks, and allow a paramter to configure the number of chunks desired. This would be a straightforward way to convert one large transaction into many smaller transactions that would fit under the lock timeout. However, MySQL does not have a method to indicate a particular range of entries in a table on which an operation should be performed. Hence this was not a possible solution. Instead, the next best way to perform this was to perform the split across the range defined by the minimum timestamp of an entry and the maximum timestamp of an entry. This is not ideal since the number of entries that bucket into a timestamp range is not equal across the timestamp ranges calculated. However with no other data field with a reasonable way to split up the transactions, this was the best option.

3.2 Refactoring

A priority during and after the bug fixing was to refactor the project heavily. Difficulties encountered when trying to learn the details of the project were caused simply due to the organization and formatting of the `translation software` obfuscating code flow and visual clarity. Hence, we primarily discuss refactoring pertaining to those categories.

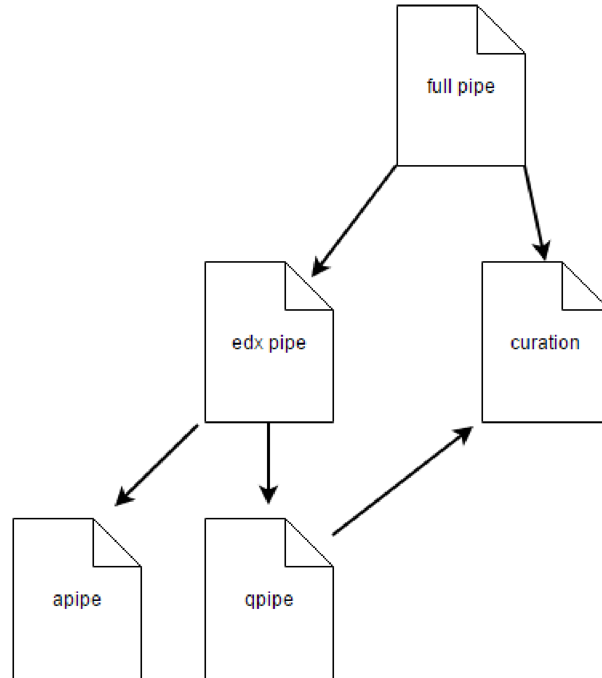

Figure 3-1: Code Organization for `translation software v1.0`

3.2.1 Code Reorganization

Originally the `translation_software` was organized in such a way that resembled the bad code organization described in Section 1.1.1 and Figure 1-2. Figure 3-1 illustrates the primary hierarchy of `translation software v1.0`. Early on in the development process the change points were identified in order to break any dependencies from lower levels of the directory tree to higher levels.

`apipe` has an unorthodox design where invoking shell scripts are the intended method to interface with the library. However, the commonly accepted practice for software projects is to import libraries directly within the source code. This required moving files in `apipe` from lower levels of the project hierarchy to higher levels, and also rewriting small parts of the code that was moved. While it is usually not desirable to stray from the organization of a library maintained by a different individual, in this case we are able to conform closer to standard software practices. Doing imports directly in source code also exposes the code flow clearly which enables code analysis tools that are useful for debugging within the text editor.

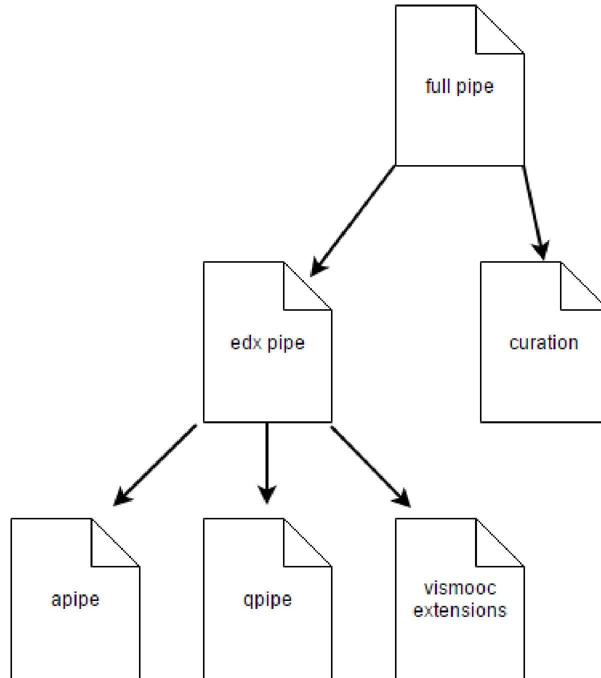

Figure 3-2: Code Organization for `translation software v1.4`

`qpipe` originally imported `curation` code which sat higher up in the project hierarchy. However given that `curation` by design is able to operate on any MOOCdb MySQL instance, as opposed to `apipe` and `qpipe` which operate specifically on Edx course data, the code that was contained in `qpipe` related to `curation` was moved up the project hierarchy which eliminated the import dependency.

The files were also reorganized such that the main user-facing file `full_pipe.py` that serves as the starting point of the project sits in the project root directory. Unused source files were also removed; code that is not reached via any code path is an often cited source of code rot because it is not kept up to date with the constantly evovling software project. With this reorganization, the project now matches that of Figure 1-1 and simplifies the code flow. Figure 3-2 illustrates the primary hierarchy of `translation software v1.4` with the new functionality added for the VisMOOC team which is further described in Section 3.3.

3.2.2 Code Reformatting

The formatting of the code throughout all modules in the `translation software` was inconsistent. This was clearly a consequence of several developers having worked on different parts of the project over its lifespan before this thesis study. However, for the continued maintenance of the project it was important to try and unify as much as possible the formatting of the code throughout the project. A purposeful choice was made not to change the code formatting in `apipe` since that is still a currently maintained project on a public facing code repository; should a future developer or maintainer of the `translation software` want to merge in changes from the public repository it is much easier if the original formatting is maintained.

However, for the remaining areas of the `translation software` that are specific to the `translation software`, the first major step taken was to run an automatic refactoring tool `yapf` [12] which is specifically a formatter for Python source files. This is a very common tool used in modern software projects which is almost effortless to use and takes a very small amount of time. However, Python in particular has code flow built into its structure. Hence, `yapf` has to be somewhat conservative, especially with regards to indentation, so as not to alter the algorithm represented by the pre-formatted code. Besides using `yapf`, minor tweaks were made by hand, such as removing superfluous whitespace.

Another important kind of reformatting performed was to remove all wildcard imports in the `translation software`. Python allows wildcard imports which consequently pollutes the namespace of a module with all the classes, methods, and global variables of an imported module, with no explicit reference to the origin of these imported classes, methods, or global variables. This is widely considered bad practice for Python code but is kept as a means of convenience for work done directly in the Python interpreter. All wildcard imports were eliminated in the spirit of removing sources of ambiguity in the codebase.

3.3 New Functionality

This section is divides the new functionality implemented for the `translation software` into two categories: functionality interally motivated within the ALFA group, and functionality externally requested by the VisMOOC team.

3.3.1 Internally Motivated

One of the earliest pieces of new functionality implemented was a small sample generation tool. The primary motivation for this was to drastically shorten the end-to-end test duration so that it would be much easier to make changes and check that the resulting MOOCdb MySQL instance either stayed the same or changed in an expected way without having to wait upwards of 10 hours (a realistic duration) for `full pipe` to finish execution or sifting through gigabytes of data. Having a short test path is crucial for developing under a TDD methodology because TDD requires quick and frequent feedback. The sample generation tool takes a raw Edx tracking log file and returns sample Edx tracking log file with one event entry for each unique event type found.

A small CFG (Control Flow Graph) generation tool was also implemented for the purpose of illustrating the code flow across the source files in the translation project. The primary motivation for developing this tool was to improve documentation on the project. The tool uses Python Call Graph [13] and Graphviz [14] to generate the CFG. Figure 3-3 shows a section of the generated CFG for the `translation software` during development.

A modification was made to the `translation software` so that upon creating a MOOCdb MySQL instance, a metadata table is also added that contains the commit hash of the translation software used to generate the MOOCdb MySQL instance. The primary motivation for this was to provide metadata for developers to help debug issues with the `translation software` should collaborators run into any issues. Attaching the commit hash to the MOOCdb MySQL instance also indicates the necessary versioning information to reproduce the same instance with the `translation`

software should someone else attempt to do so. Previously the notion of versioning within the translation software was non-existent so this was a push towards explicitly defined versions, which also help delineate changes clearly between revisions of the translation software documented by changelog notes.

3.3.2 Externally Requested

The motivation for all of the following added functionality is to interface MOOCdb with VisMOOC. Previously existing documentation was inaccurate in many places and hence inadequate for bridging the gap between MOOCdb and VisMOOC. Figure 3-4 and Figure 3-5 show the MOOCdb v1.4 schema based on the output of translation software v1.4. The following paragraphs describe the added tables.

The first request we were given from the VisMOOC team was to extend the functionality of the translation software so that it could create and populate a table of Edx course click events. This required changes to the MOOCdb schema as well as implementing a new module in qpipe named clickevents that would be responsible for recording all click events found in the intermediate csv. Afterwards it was also requested from us that we do the same for user enrollment events, which would populate a course users table. This too is implemented as a new module in qpipe named courseusers.

The next request was to provide functionality that would enable an overall view of users across several MOOCdb courses, each contained in a separate MOOCdb MySQL instance on the same host machine. Similarly to clickevents and courseusers, this functionality is implemented as a new module in qpipe named userids. However, it must check almost every Edx event for user information unlike the clickevents and courseusers module which only check a much smaller set of events. The user ids are imported into a special MySQL database named moocdb_index. The intended use of the moocdb_index is for a developer or scientist to make a query for a particular hashed user id[1] which will then return the names of the MOOCdb MySQL instances

[1] The original design of MOOCdb placed a lot of emphasis on privacy and hence all Edx usernames are hashed during the apipe stage of the translation software with a RIPEMD160 40 char

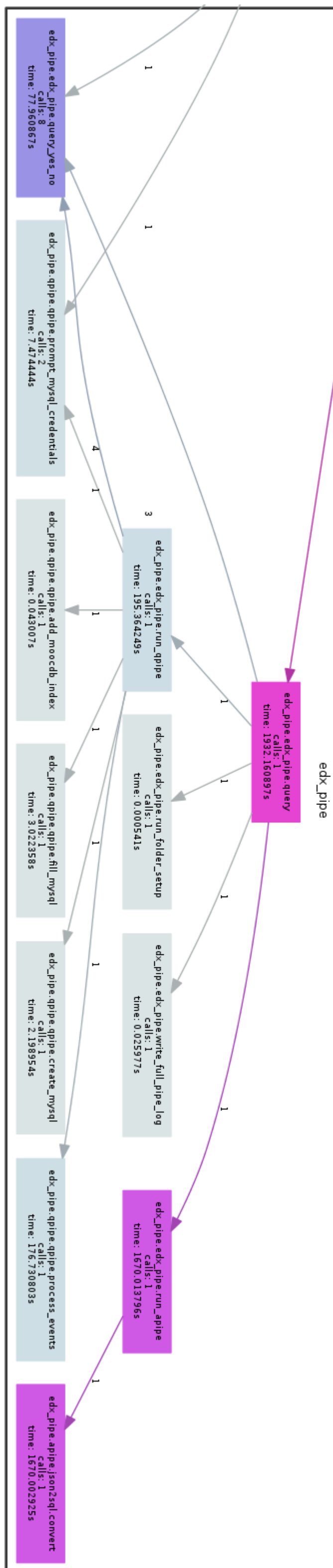

Figure 3-3: Portion of `full pipe` Control Flow Graph

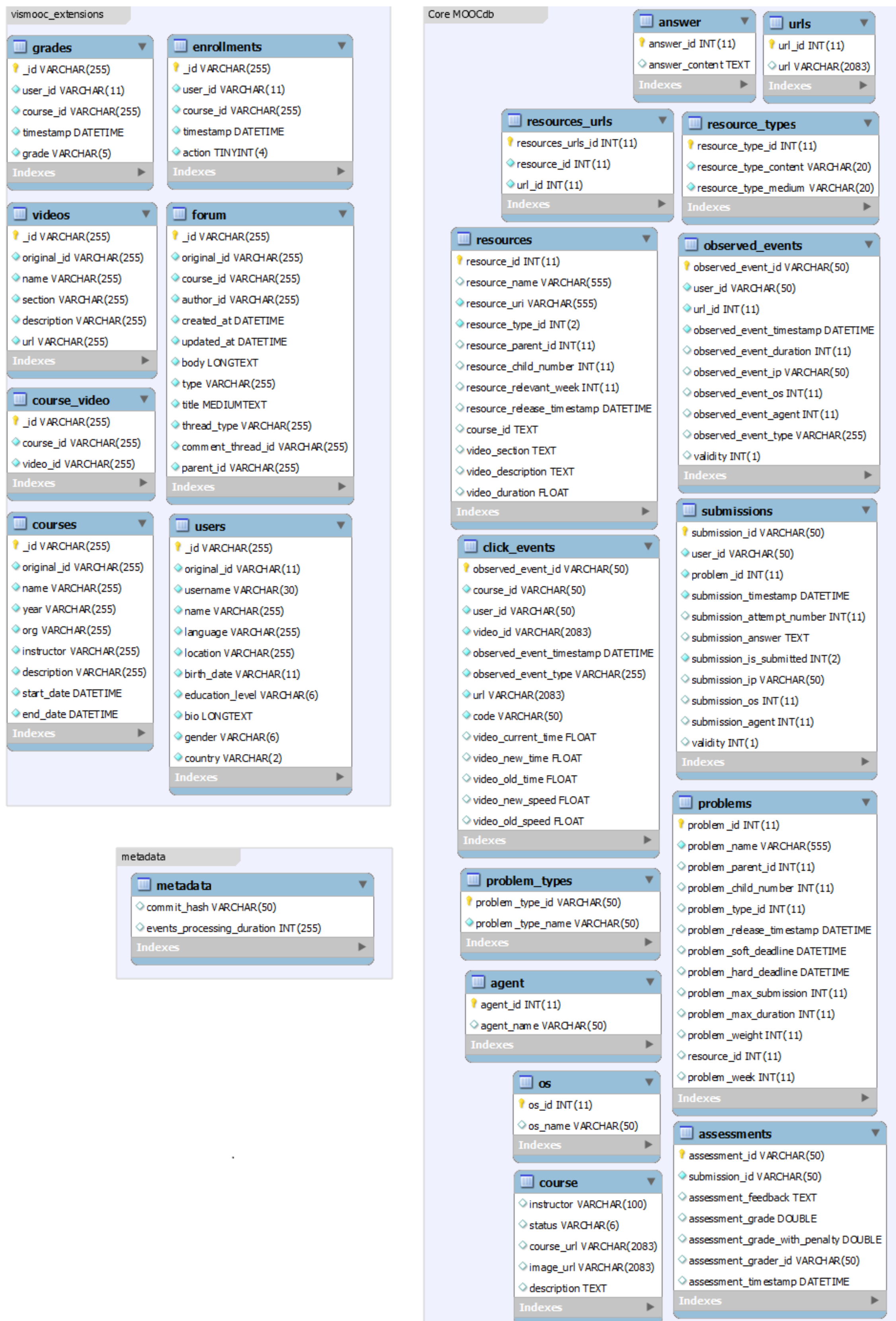

Figure 3-4: Core MOOCdb `v1.4` and `vismooc_extensions` tables

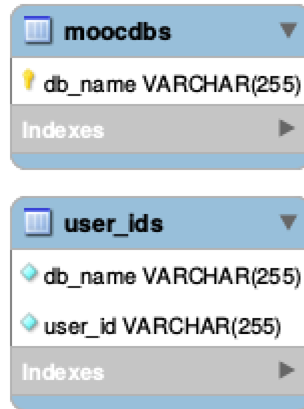

Figure 3-5: `moocdb_index` tables

```
mysql> select * from moocdb_index.user_ids where user_id = "fedb32d7c2c25fc4bf91d696b511d15211537991";
+------------+------------------------------------------+
| db_name    | user_id                                  |
+------------+------------------------------------------+
| 201_sample | fedb32d7c2c25fc4bf91d696b511d15211537991 |
| 203_sample | fedb32d7c2c25fc4bf91d696b511d15211537991 |
+------------+------------------------------------------+
```

Figure 3-6: `moocdb_index` query for a user in more than one course

on the host machine containing data associated with the hashed user id. Figure 3-6 shows an example of such a query. Using this, other applications interfacing with the MOOCdb host machine can create a view of a particular user over several MOOCdb courses.

Since MOOCdb's ultimate goal is to act as a unifying MOOC data format, there will inevitably be shortcomings that different third party groups will want to address via their respective modifications to the `translation software`. We introduced the concept of extensions as our choice for providing separation from the core `translation software` logic while still being easy to interface with the `translation software`.

The next set of requests from the VisMOOC team we deemed to be outside of the core MOOCdb `translation software` which takes an Edx tracking log file as input and outputs a MOOCdb course MySQL instance. The VisMOOC team wanted

hash. It is important for any data associated with usernames outside of the Edx tracking log to be hashed the same way before importing such data into a MOOCdb MySQL instance to maintain compatability.

more tables to be added to the schema to interface with the VisMOOC software but unlike `clickevents` and `courseusers`, the data with which they wanted to populate the new tables did not come from the Edx tracking log file. Instead, they came from other files in the Edx course data that is packaged together. Hence, our solution to this was to create a new directory on the same level as `apipe` and `qpipe` called `vismooc_extensions` for modules that would implement vismooc specific functionality. We decided this because the requests were getting more specific to VisMOOC software requirements and straying from the original goal of the MOOCdb project which is to create a sufficiently generic MOOC schema that should encapsulate the most important kinds of data from different MOOC hosts. In fact, one of the non-tracking log Edx course files they wanted to pull data from were not found in any of our Edx course data files. For this particular file that we did not have, based off a sample of the file from the VisMOOC team we implemented extensions modules called `courses`, `course_video`, `videos`, `grades`, `enrollments`, `forum`, and `users` under the `vismooc_extensions` directory that would implement the tables they requested, and then wrote up general instructions of how further modules can be implemented and where the change points are located in the `translation software`.

The aforementioned additions are illustrated in the `full pipe v1.4` flowchart diagrammed in Figure 3-7, extending onto the `full pipe v1.0` flowchart diagrammed in Figure 2-2.

3.4 Test Harness

During the implementations of new functionality, a test harness was also being developed alongside it. In the spirit of conforming to modern software practices, the new functionality described in Section 3.3.2 was implented using TDD (Figure 1-5). Each module implemented for the VisMOOC team was accompanied with the development of tests. Each test was placed in the same directory as the corresponding module with `_test` appended to the filename. This was done first because it was code written fresh for the translation software and so it was more familiar and easier to write tests.

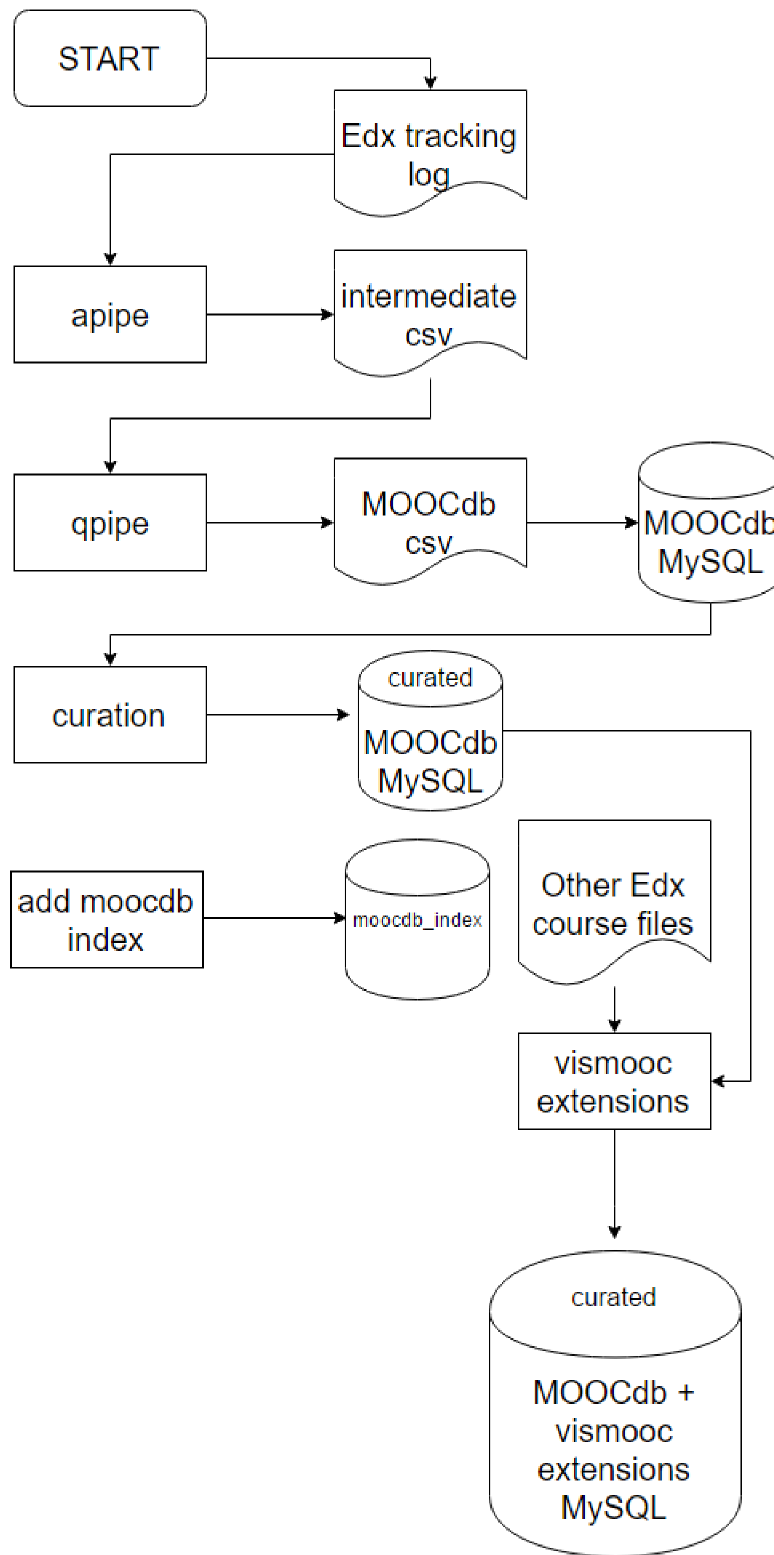

Figure 3-7: `full pipe v1.4` Flowchart

Similarly to the tests written for the modules requested by the VisMOOC team, the tests written for the legacy code were placed in the same `qpipe` directory with `_test` appended to the filename. The goal was for every identified critical module, reach a minimum baseline of testing every function that was not a helper function only used by the module within which it was contained. The VisMOOC extensions were developed using TDD, keeping with the spirit of maintaining good software practices.

There is no end-to-end minimum test because the primary user-facing module `full_pipe.py` is designed in a way that requires user interaction. Since the user is prompted for MySQL login credentials that will likely differ between different hosts, it was decided that in place of an automated unit tests a tutorial would be written as the minimum end-to-end test.

3.5 Documentation

While developer-facing documentation was touched up and updated here and there during the early stages of taking over the `translation software`, method docstrings were primarily updated as necessary during the development of tests for the legacy code. The tests themselves also constitute a form of documentation as they fundamentally represent a mapping from inputs to outputs.

The most significant form of documentation provided was through step by step guided tutorials written for our collaborators on the VisMOOC team. The tutorials include all the details on any configuration necessary and the terminal commands used to run the `translation software`. The expected output is also included as well in order to provide confidence to collaborators that the observed behavior of the `translation software` is correct. There are tutorials written for the main `translation software` and the `moocdb index` functionality. These tutorials provide a very friendly starting point for our collaborators to learn the `translation software` source code. Coupled with the refactoring that transformed the codebase into a tree hierarchy, it is designed to be easy to determine the code paths of the `translation software`.

Chapter 4

Evaluation of `translation software v1.4`

In this section we discuss our evaluation of the applicaton of LCCA on the `translation software` for MOOCdb. We evaluate the success of applying such a method using criteria of understandability, resistance to change, stability, and result reproducibility. A description of these criteria can be found in Section 2.1.

4.1 Understandability

As mentioned previously in Section 3.2.1, all upwards dependencies in the file hiearchy were refactored out. There were 2 such dependencies (one in `apipe` and one in `qpipe`) in code sections still being used in the current version of the project. Another dependency was located in a file that was mostly an unused duplicate of the main `qpipe.py` file, and was deleted entirely instead. Additionally, the code was reformatted to remove Python wildcard imports (Section 3.2.2) as well. 11 wildcard imports were removed from files that are still used in the recent versions of the `translation software`. There were wildcard imports in files that eventually were deleted. Unused code was also deleted as it was both not relevant since it was not being used, and had no maintainer or corresponding documentation. Most of the deleted code resided in an unused `coursera` module and unused `curation` source files. The amount of

deleted lines from source files amount to approximately 5k. We were not concerned with deleting all of this code because source code version control allows us to easily retrieve the deleted files should we decide to revisit them in the future. These refactors made the code flow much more simpler and explicit, thus improving understandability directly.

We use `pylint` scores to get a quantative estimate of readability (and hence understandability). `pylint` is a static code checker for Python that analyzes code for violations of defined style guides. We ran `pylint` on `v1.0` and `v1.4.4` of `translation software` to get a comparison of readability.[1] Table 4.1 summarizes the score differences between selected modules from `translation software v1.0` and `translation software v1.4.4`. The huge improvement from `extractor` was a consequence of eliminating non-standard Python class attribution setters and getters. The small improvement from `genformatting` was a consequence of the module being a low priority target for cleanup. Of note 9 modules have changed from negative to positive scores.

Separate plots for `qpipe` and `curation` modules have also been plotted as bar charts to compare `v1.0` and `v1.4.4` in Figures 4-1 and 4-2 respectively. Only modules that existed in both `v1.0` and `v1.4.4` have their own rows in the table. The overall score computation is an aggregate across all modules in the respective versions regardless of whether or not the other version contains those source files, with all `apipe` source files excluded. The upper bound for the `pylint` score is 10 whereas but is no lower bound. A score of 5 is okay for undocumented code and a score of 7 is a decent minimum baseline to target. As indicated by the table, `v1.4.4` scored much higher than `v1.0` across the board, with fairly large score differentials for most of the individual modules. This gives us high confidence that `v1.4.4` is much more readable than `v1.0`.

---

[1] `qipe/qpipe.py` was `qpipe/main.py`, and `curation/curation.py` was `curation/main.py` in `translation software v1.0`. `pylint v1.6.4` and `Python 2.7.6` were used in computing the `pylint` scores. See Section 6.1 of [15] for the definition of the `pylint` score.

Module	v1.0	v1.4.4	Δ
qpipe/events.py	-0.98	9.69	10.67
qpipe/eventformatter.py	2.46	8.78	6.32
qpipe/extractor.py	-13.33	7.50	**20.83**
qpipe/genformatting.py	5.00	6.58	**1.58**
qpipe/helperclasses.py	2.61	8.20	5.59
qpipe/moocdb.py	-2.92	9.60	12.52
qpipe/inheritloc.py	4.44	9.44	5.00
qpipe/resources.py	2.31	7.86	5.55
qpipe/specformatting.py	5.00	6.82	1.82
qpipe/submissions.py	-5.20	7.62	12.82
qpipe/util.py	3.83	5.71	1.88
qpipe/updateloc.py	3.89	10.00	6.11
qpipe/qpipe.py	-2.54	8.62	11.16
curation/curation.py	-2.41	7.27	9.68
curation/submissions_curation.py	-2.88	9.15	12.03
curation/sql_functions.py	-3.22	8.61	11.83
curation/observed_events.py	-3.61	8.97	12.58
full_pipe.py	1.83	8.48	6.65
Overall Score	-1.27	8.79	10.06

Table 4.1: pylint Score Comparison for translation software

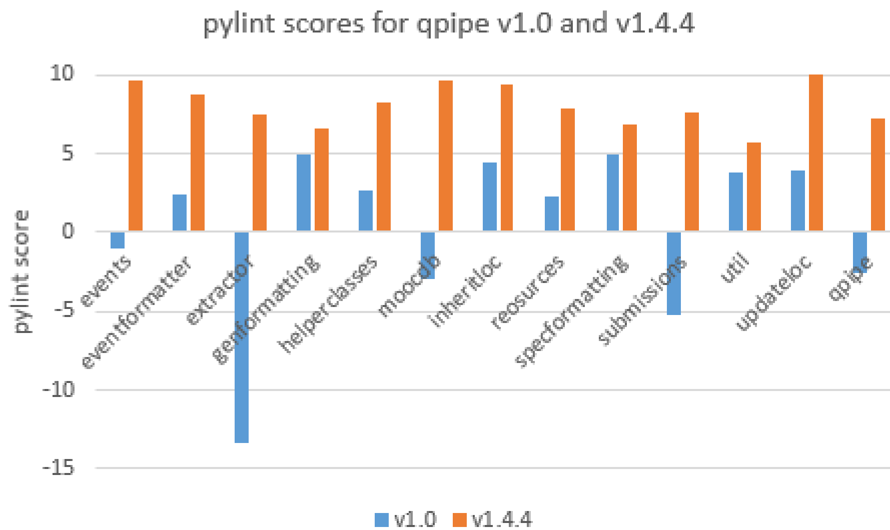

Figure 4-1: pylint scores for qpipe v1.0 and v1.4.4

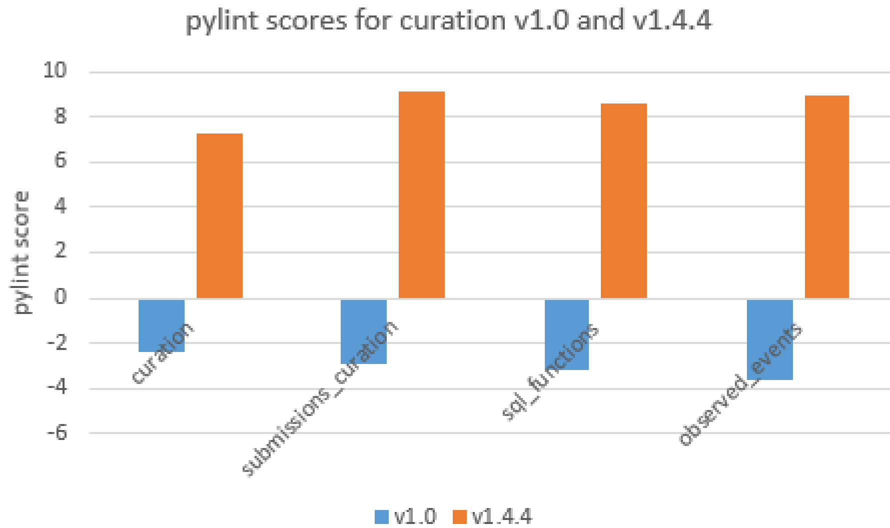

Figure 4-2: `pylint` scores for `curation` `v1.0` and `v1.4.4`

4.2 Resistance To Change

We aim to evaluate the newer versions of the `translation software` on their resistance to change. Figure 4-3 illustrates the commit activity during this thesis study. The peaks around September and October were artificially inflated due to the `README` file being used as an issue tracker during that date range. The peak mid-November is similarly inflated by commits made to add higher level documentation, but not source code documentation, for our VisMOOC collaborators before an in-person meeting with them. However around this same time both the major refactoring changes (Section 3.2) and the sample generation tool (Section 3.3.1) were implemented. The period from mid-September to mid-November should thus be considered as largely inactive with regards to meaningful commits (i.e. commits that modify source code). However now you can see that after mid-November there are many lower and wider regions of commit activity, which are indeed entirely source code changes. The `translation software` experienced much more frequent meaningful activity after mid-November compared that of the period before mid-November. This illustrates the strongly positive effects of the sample generation and major refactoring on the `translation software`'s adaptability.

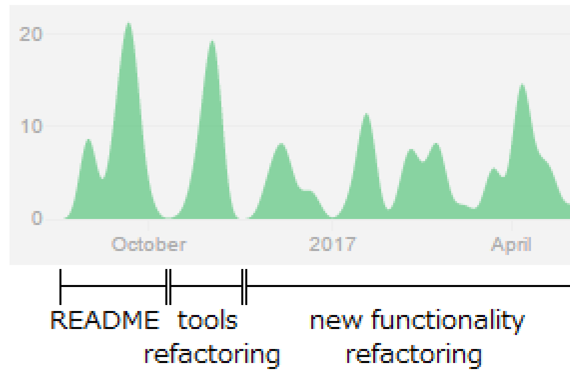

Figure 4-3: Number of Commits v. Date

Furthermore, given the back and forth with the VisMOOC team from versions `v1.4.0` to `v1.4.4`, which was developed over a fairly tight 1 to 2 week period, we think that the current version of the `translation software` has a much lower resistance to change compared to `v1.0`. `v1.0` was littered with numerous syntax error or unclear requirements for input filenames and input directory structure. After combing through the code to resolve those issues just to get the code to run, it took a few weeks before an actual end-to-end execution of `translation software v1.0` could even be reached. At this point, even though `translation software v1.0` could be execute it took a whole day to actually complete because it would process an entire semester of Edx course data. Hence it took a whole day just to even run `v1.0` to see if changes worked as intended. On the other hand, the sample generation tool and fixes of `v1.4` enabled much quicker and bug-free end-to-end executions. The quicker executions finish in less than a minute on generated samples and new bugs can be detected and squashed in a matter of minutes as a result of this much shorter development cycle.

4.3 Stability

A strong test harness is necessary for ensuring that a software project adheres to the expected outputs for a particular set of inputs. `qpipe` now has 9 module tests with 54 unit tests spread across them. `vismooc_extensions` has 6 module tests with 56 unit tests spread across them. Table 4.2 shows the distribution of tests across the modules.

Module	Number of Tests
qpipe/clickevents_test.py	8
qpipe/eventformatter_test.py	12
qpipe/eventmanager_test.py	1
qpipe/events_test.py	16
qpipe/genformatting_test.py	6
qpipe/inheritloc_test.py	3
qpipe/specformatting_test.py	3
qpipe/updateloc_test.py	3
qpipe/userids_test.py	2
vismooc_extensions/courses_test.py	12
vismooc_extensions/enrollments_test.py	6
vismooc_extensions/forum_test.py	12
vismooc_extensions/grades_test.py	6
vismooc_extensions/users_test.py	13
vismooc_extensions/videos_test.py	7
Total Tests	110

Table 4.2: qpipe and vismooc_extensions Test Count

There is no comparison to v1.0 because all of its tests either had syntax errors that prevented it from executing, or did not have any assertions to automatically check output. Based on brief comments it appears that many of the tests were designed in such a way that a developer just observed console output to see if it looked reasonable. Hence v1.0 had effectively 0 unit tests.

These unit tests helped ensure the translation software maintained its current level of correctness. There was an incident where a revision was pushed to the remote repository that was not checked across all existing tests and another person caught it by running the existing test suite. This prevented a bug quickly and early before it could potentially get out of hand down the road.

We used pytest [16] to generate coverage reports.[2] We use the coverage reports to get an estimate of how much of the translation software code is actually being tested by the test harness developed over the thesis study. Coverage reports were not generated for apipe because that is an open source library and we defer to the current developer and maintainer to maintain his own unit tests. We were also not

[2]The command "pytest -cov-report html:cov_html -cov=. ." was executed in the qpipe and vismooc_extensions directories for v1.4.4 to generate the respective coverage reports.

able to figure out how to run the tests that were included with the library.

Table 4.3 shows a 62% coverage overall for qpipe which is decent, but leaves plenty of room for improvement. Table 4.4 shows a 84% coverage overall for vismooc_extensions which is even better. The disparity between the two can be attributed to the fact that qpipe was mostly developed strictly under LCCA when creating its test harness, and not all of the modules were covered due to time constraints. However, vismooc_extensions was largely developed under TDD which meant writing the tests first before writing actual source code to implement functionality. The test data was created by taking the minimum data necessary to reproduce an equivalent JSON object or table row to test that the vismooc_extensions modules were performing the mapping from the Edx data to the vismooc_extensions tables correctly. Any personally identifying information from the test samples has been removed.[3]

We also plotted the coverage data by grouping tests with their corresponding modules and removing configuration files and Python __init__.py (i.e. empty) files from consideration.

The coverage data is plotted for the qpipe directory in Figures 4-4 and 4-5. The cluster of points in the top left of Figure 4-4 can be attributed to the fact that applyling LCCA to existing legacy code was simpler to do for the modules that contained less statements total. The cluster of points in the top left of Figure 4-5 has a simpler explanation: fewer missing statements corresponds to greater coverage. The data points with 0% coverage are a consequence of the lower priority modules not making the cut for our deadlines when deciding where to add tests.

Similarly the coverage data is plotted for the vismooc_extensions in Figures 4-6 and 4-7. These plots are different from Figures 4-4 and 4-5 in that there is no such clustering behavior present. This is because this was new code developed mostly using TDD which requires tests to be written before implementation. The one data

[3]However, one mapping from the Edx profile course file to the users table is a **blatant violation of privacy**. In the users module, the real name of the user is preserved without anonymization. While the test samples have no personally identifying information, actual Edx course data will have such personally identifying information. This mapping will likely be changed in the near future but completing the interface with the VisMOOC software was prioritized. A note has been made in the source code of the offending section.

Module	statements	missing	coverage
__init__.py	0	0	100%
clickevents.py	9	1	89%
clickevents_test.py	51	2	96%
config.py	25	0	100%
eventformatter.py	82	37	55%
eventformatter_test.py	81	2	98%
eventmanager.py	25	7	72%
eventmanager_test.py	20	2	90%
events.py	98	6	94%
events_test.py	148	12	92%
extractor.py	40	40	0%
genformatting.py	38	3	92%
genformatting_test.py	46	2	96%
helperclasses.py	283	107	62%
inheritloc.py	18	1	94%
inheritloc_test.py	24	2	92%
moocdb.py	25	2	88%
qpipe.py	145	145	0%
resources.py	182	182	0%
specformatting.py	22	1	95%
specformatting_test.py	37	2	95%
submissions.py	21	21	0%
updateloc.py	18	1	94%
updateloc_test.py	19	2	89%
userids.py	12	1	92%
userids_test.py	56	2	96%
util.py	7	0	100%
Total	1532	584	62%

Table 4.3: `qpipe v1.4.4` Test Coverage

Module	statements	missing	coverage
`__init__.py`	0	0	100%
`config.py`	6	6	0%
`courses.py`	62	10	84%
`courses_test.py`	53	2	96%
`enrollments.py`	5	0	100%
`enrollments_test.py`	24	2	92%
`forum.py`	39	5	87%
`forum_test.py`	62	2	97%
`grades.py`	5	0	100%
`grades_test.py`	24	2	92%
`users.py`	33	18	45%
`users_test.py`	46	2	96%
`util.py`	24	13	46%
`videos.py`	49	10	80%
`videos_test.py`	44	4	91%
`Total`	476	76	84%

Table 4.4: `vismooc_extensions v1.4.4` Test Coverage

point with the lowest coverage under 50% is a utility module that is used by the other modules that slipped through the cracks due to deadlines.

4.4 Reproducibility

Reproducibility was an important criteria for us in academia since it is normally important for a third party to be able to reproduce the result of a paper. Ideally given the inputs and code, a third party can reproduce the expected outputs of a study. This becomes increasingly relevant as software continues to be used more in academia across different disciplines than before. That being noted, releasing a demo or code is not even a requirement for many Computer Systems journals and conferences [17]. This seems to fly in the face of the long held emphasis of reproducibility of results for many academic disciplines. Reproducibility is also crucial for debugging errors that were not covered by an existing test suite. Hence, our stance is that reproducibility is still important means of supporting the claims of academic work, and that is why the metadata table was added (Section 3.3.1) to tag MOOCdb instances generated

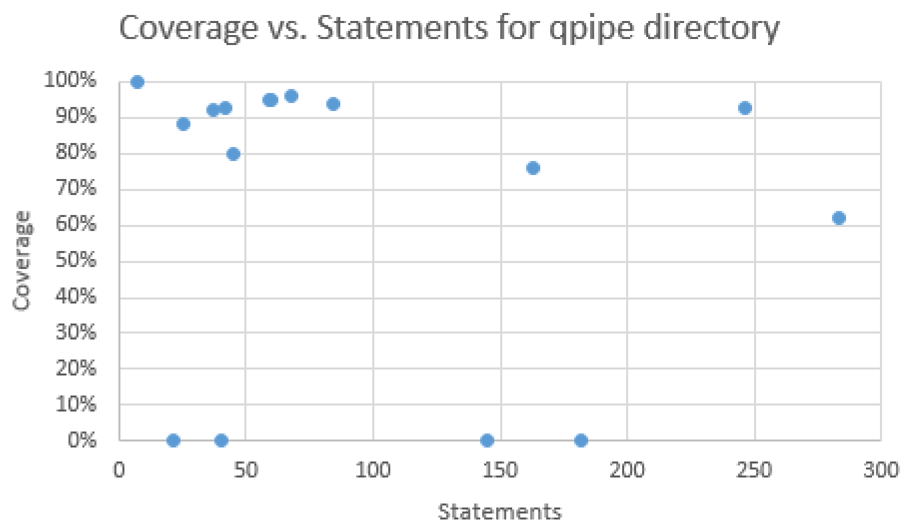

Figure 4-4: Scatterplot of Coverage vs. Statements for `qpipe v1.4.4`

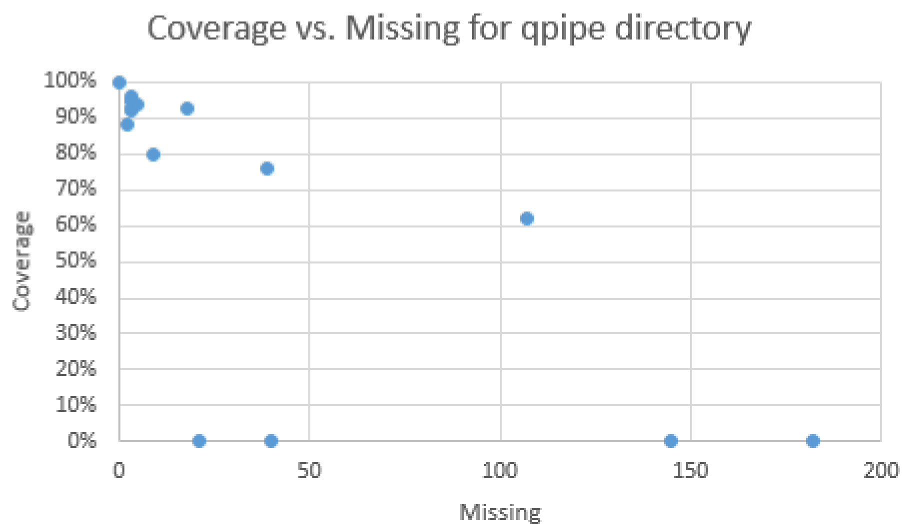

Figure 4-5: Scatterplot of Coverage vs. Missing for `qpipe v1.4.4`

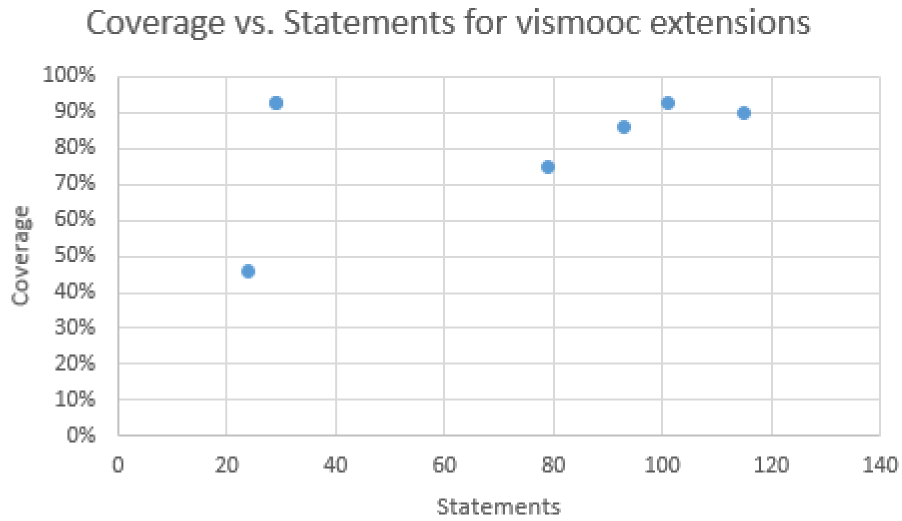

Figure 4-6: Scatterplot of Coverage vs. Statements for `vismooc_extensions v1.4.4`

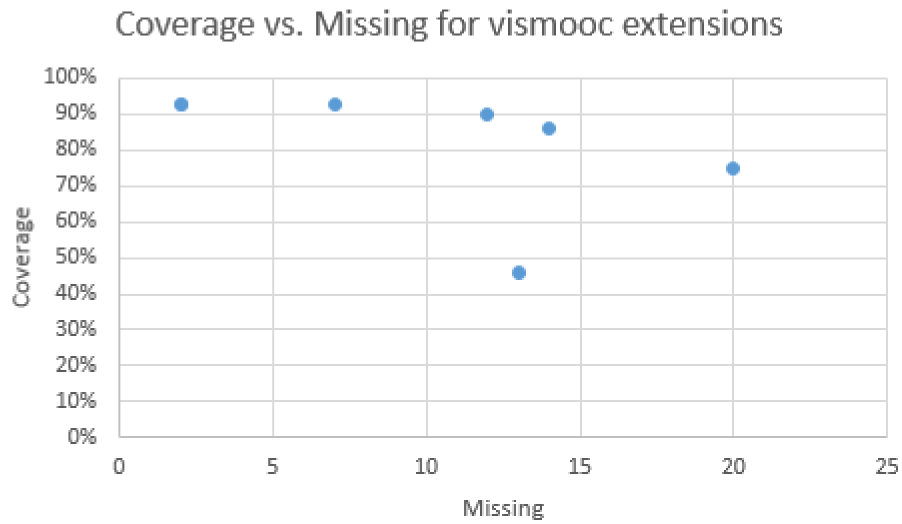

Figure 4-7: Scatterplot of Coverage vs. Missing for `vismooc_extensions v1.4.4`

with the `translation software` with the commit hash being used. The commit hash provides a very explicit means of versioning the software used. This can be referenced in academic work when describing the measurement process and would reduce ambiguities that would arise otherwise since changes between different versions of software can lead to different outcomes. Furthermore, a major use case for this data is analytics which requires high confidence in the integrity and any knowledge of the treatment and manipulation of such data.

During collaboration towards the end of this study the VisMOOC team found that our `v1.4.0` release which implemented the prototype of the `vismooc_extensions` containing an implementation of the `courses` module actually crashed on `qpipe` and `curation` on different Edx courses' respective tracking logs. The VisMOOC team sent us console output error logs and sample inputs that created the error and we were able to successfully reproduce the exact same error logs on our side. Once we were able to do this, tracking down the source of the bug became much quicker. We faced a similar situation with our `v1.4.3` release except instead of a bug that crashed the `translation software`, the output in the MySQL MOOCdb instance was not as expected from the VisMOOC team. In the same manner as before, they sent us a sample of erroneous output and the input data samples that were necessary to produce the same erroneus output. Tracking down the source of the problem after that point just required some brief searching through the sample data input file, and soon after the problem was resolved. The two aforementioned cases are promising results suggesting that the newer versions of the `translation software` are much more capable of reproducing outputs consistently, even on different host machine environments.

Chapter 5

Conclusion

This thesis has presented an application of LCCA on a legacy software project developed originally within our research group. We find that by following LCCA, we were able to improve the understandability, stabilty, and reproducibility of the MOOCdb `translation software` while also reducing its resistance to change. We see the overall `pylint` score of the `translation software` go from -1.27 to 8.79 from `v1.0` to `v1.4.4`. We also see the test count essentially go from 0 to 110 from `v1.0` to `v1.4.4`. We also find that the test coverage for `qpipe` and `vismooc_extensions` is 62% and 84% respectively, a significant improvement from its previous 0%. These statistics give us confidence that the `translation software` has indeed improved noticeably according to our evaluation focuses.

One big lesson we learned is that automation is key for keeping code clean. Using automatic code refactoring to shift the inconsistent visual style throughout the codebase to a consistent one was much quicker and prone to less errors than trying to perform such a refactoring manually. Also, the test harness itself provides automatic verification upon simply executing the test itself. There is no room for human error on the part of manually observing the output. Since automation plays such a big factor keeping the code clean, we highly recommend the same use of automation is adopted by future maintainers of the `translation software` to avoid relapse into an ugly codebase like that of `v1.0`.

Besides that, adherence to TDD during LCCA when possible was also a big factor

in cleaning up the code. The higher coverage score for `vismooc_extensions` was a result of following TDD for most of the development of that code. Future maintainers of the `translation software` are highly encouraged to follow TDD as much as possible, and otherwise use LCCA if they cannot follow TDD.

Chapter 6

Future Work

The chapter details potential avenues of further study that could not fit in the time constraints of this thesis study.

6.1 Continuous Integration

In the early stages of the study continuous integration was proposed as a tool that would be very useful in enforcing clean code standards. Continuous integration is a system in which a remote server monitors a code repository for changes and then runs the entire test suite on every revision of the project that is pushed to the repository. If any test fails then the pushed commit is marked with the failure and an error log is generated with the details of the failing test. This sort of tool takes the discipline out of the hands of the developer in ensuring the tests always still pass. In fact, during this thesis study there was an instance of a revision that was pushed to the remote repository that failed a test and it was only due to another person executing the tests independently that this was caught (Section 4.3). Continuous integration would leave no room for this kind of error to slip by.

6.2 Automatically Generated Documentation

Another idea proposed in the early stages of the study was the idea of automatically generating documentation. Manually written documentation is prone to going out of date, especially with quick cycle development methods such as TDD. At the end of the day, the code is the absolute authority on describing what code does. The basic idea of automatically generated documentation was to write an external script to parse design areas of the code that were expected to remain stable with regards to formatting and extract useful information that should be exposed to the developers without requiring them to dig into the code. An example of how this may apply to the `translation software` project is the MOOCdb schema which throughout this thesis study has undergone several changes. The MOOCdb schema is a crucial piece of information and so it is required that collaborators understand clearly what it is for any particular revision of the `translation software`. Our collaborators actually ran into difficulties trying to understand what data was actually contained in the converted MOOCdb courses from the `translation software` project due to a woefully out of date wiki page [18]. Automatically generated documentation could have prevented many of these headaches. The schema for `vismooc_extensions` of `translation software v1.4` in the meantime can be found in the README file of the `vismooc_extensions` directory. A user-facing documentation for this schema has not been generated yet because there are likely changes that will still take place in the near future, such as tables that were initially created for the VisMOOC team becoming obsolete by `v1.4`. Since changes are likely to take place this information should only be exposed to developers until a stable state has been reached.

Bibliography

[1] Robert Cecil Martin. (2008) *Clean Code: A Handbook of Agile Software Craftsmanship.*

[2] Zachary A. Pardos, Kevin Kao. moocRP: An Open-source Analytics Platform. In *Learning@Scale*, 2015.

[3] http://agiledata.org/essays/tdd.html

[4] http://www.agile-process.org/change.html

[5] Mohit Rajput. (2017) https://dev.to/mohitrajput987/importance-of-writing-clean-code.

[6] https://academia.stackexchange.com/questions/17781/why-do-many-talented-scientists-write-horrible-software

[7] Michael C. Feathers. (2004) *Working Effectively with Legacy Code.*

[8] MOOCdb. http://moocdb.csail.mit.edu

[9] Andreas Paepcke. json_to_relation. https://github.com/paepcke/json_to_relation

[10] Quentin Agren. *From clickstreams to learner trajectories.* Masters thesis for École Normale Supérieure de Lyon, 2014.

[11] VisMOOC. http://ihome.ust.hk/~qchenah/vismooc/.

[12] yapf. https://github.com/google/yapf.

[13] Python Call Graph. http://pycallgraph.slowchop.com/en/master/

[14] Graphviz. http://www.graphviz.org/

[15] https://docs.pylint.org/en/1.6.0/faq.html

[16] https://docs.pytest.org/en/latest/

[17] https://academia.stackexchange.com/questions/23237/why-are-papers-without-code-but-with-results-accepted

[18] http://moocdb.csail.mit.edu/wiki/index.php?title=MOOCdb