Code Collusion Checker

Final Report for CS39440 Major Project

Author: Stephen John Norwood (sjn3@aber.ac.uk)
Supervisor: Chris Loftus (cwl@aber.ac.uk)

8th May 2017
Version: 1.0 (Release)

This report was submitted as partial fulfilment of a BSc degree in Computer Science (inc Integrated Industrial and Professional Training (G401)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name Stephen Norwood

Date 21/04/2017

Consent to share this work

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name Stephen Norwood

Date 21/04/2017

Acknowledgements

I am grateful to Chris Loftus for his support during the project, and for the weekly meetings that helped keep the project on track. I am also grateful to Lawrence Tyler for his feedback on the mid-project demonstration, which was taken on board and pushed the project in the right direction.

I'd like to thank my parents, and Alice Doody for proof reading and providing overall feedback on the final report.

Abstract

The Code Collusion Checker is an application designed to check for collusion between code submitted from different students for the same assignment. To be used by a lecturer, the application is hosted as a web application with a separate Java backend, to complete the processing of the code while not inhibiting the performance of the website.

Currently the only solution to the issue of checking for collusion between students is to use the Measure of Software Similarity (MOSS) tool developed by Stanford University (CITE). The Code Collusion Checker has been developed to be user friendly, and to be usable without any requirements. By having the point of access of the application as a website, it is accessible on multiple devices and platforms, without any client-side requirements apart from an internet connection.

Checking of code is to be done on two levels, String Matching and Code Parsing. In the prototype delivered in this project, only String matching has been completed, with research and discussion on Code Parsing.

CONTENTS

1 Background & Objectives **1**
- 1.1 Background . . . 1
  - 1.1.1 Cloud providers . . . 1
  - 1.1.2 Programming languages and tools . . . 2
  - 1.1.3 Stanford University's Measure of Software Similarity . . . 4
- 1.2 Process . . . 4
- 1.3 Analysis . . . 5
  - 1.3.1 Cloud Computing . . . 5
  - 1.3.2 Security . . . 5
  - 1.3.3 Feature List . . . 6

2 Design & Implementation **8**
- 2.1 Iteration 0 . . . 8
  - 2.1.1 Initial design . . . 8
  - 2.1.2 Initial setup . . . 13
- 2.2 Iteration 1 . . . 14
  - 2.2.1 Upfront design . . . 14
  - 2.2.2 Feature #4: Uploading single submissions . . . 14
  - 2.2.3 Retrospective . . . 16
- 2.3 Iteration 2 . . . 17
  - 2.3.1 Upfront design . . . 17
  - 2.3.2 Feature #2: Creating assignments . . . 18
  - 2.3.3 Feature #8: Reviewing results of an assignment . . . 18
  - 2.3.4 Retrospective . . . 18
- 2.4 Iteration 3 . . . 19
  - 2.4.1 Upfront design . . . 19
  - 2.4.2 Feature #6: Checking submissions via string matching . . . 20
  - 2.4.3 AWS Lambda . . . 21
  - 2.4.4 Retrospective . . . 21
- 2.5 Iteration 4 . . . 22
  - 2.5.1 Upfront design . . . 22
  - 2.5.2 Feature #1: User Accounts . . . 23
  - 2.5.3 Retrospective . . . 24
- 2.6 Iteration 5 . . . 25
  - 2.6.1 Upfront Design . . . 25
  - 2.6.2 Feature #12: Worker Configuration File . . . 25
  - 2.6.3 Feature #14: Enhancement of String matching . . . 25
  - 2.6.4 Retrospective . . . 26
- 2.7 Iteration 6 . . . 27
  - 2.7.1 Upfront design . . . 27
  - 2.7.2 Feature #3: Uploading and using templates . . . 28
  - 2.7.3 Feature #5: Uploading multiple submissions . . . 29
  - 2.7.4 Feature #10: Uploading submissions with multiple files . . . 29
  - 2.7.5 Research into parse checkers . . . 30
  - 2.7.6 Retrospective . . . 30

2.8 Iteration 7 . . . 31
2.8.1 Feature #9: Checking submissions via code parsing . . . 31
2.8.2 Testing . . . 31

3 Testing **32**
3.1 Overall approach to testing . . . 32
3.2 Automated testing . . . 32
3.2.1 Unit tests . . . 32
3.2.2 Cucumber testing . . . 33
3.3 Manual testing . . . 33
3.3.1 Iterative . . . 33
3.3.2 Stress testing . . . 34
3.3.3 Comparing other software . . . 34

4 Evaluation **36**
4.1 Project requirements and feature comparison . . . 36
4.2 Methodology . . . 37
4.3 Design decisions . . . 37
4.4 Tools and technologies used . . . 38
4.5 Final product . . . 39
4.6 Future Work . . . 39
4.7 Summary . . . 40

A Third-Party Code and Libraries **41**
1.1 Libraries . . . 41
1.2 Code . . . 41
1.2.1 Levenshtein Distance . . . 41

B Ethics Submission **44**

C Code Examples **46**
3.1 Custom Levenshtein Distance implementation . . . 46
3.2 Reading zip files from S3 . . . 50

Annotated Bibliography **53**

LIST OF FIGURES

1.1 The feature list created as part of the FDD-like methodology 7

2.1 The system-level sequence diagram created as part of the overall model 9
2.2 The system-level use case diagram created as part of the overall model 10
2.3 The initial entity relationship diagram for the database 10
2.4 The initial class diagram for the MVC website 11
2.5 The initial class diagram for the Java back-end worker 12
2.6 Iteration 2 Entity-Relationship diagram introducing columns to assignments and submissions 17
2.7 Iteration 2 website class diagram adding new controllers, and Submission model methods 17
2.8 Iteration 3 worker class diagram introducing factories and interfaces with methods 19
2.9 Matrix showing path of Levenshtein Distance [1] 20
2.10 Iteration 4 worker class diagram introducing singleton managers 22
2.11 Iteration 4 website class diagram introducing users and sessions 23
2.12 Iteration 4 database entity-relationship diagram updating the users table 23
2.13 Iteration 5 worker class diagram adding the ConfigurationManager 25
2.14 Iteration 6 database entity-relationship diagram adding the results table and updating assignments 27
2.15 Iteration 6 website class diagram adding template method to assignments model . 27
2.16 Iteration 6 worker class diagram updating to accommodate for templates and multi-file projects 28

3.1 Results produced by the MOSS system. 35

Chapter 1

Background & Objectives

1.1 Background

To prepare for the project, research was put into the following areas:

- Cloud Providers
- Programming Languages and Tools
- Stanford University's Measure of Software Similarity (MOSS)

1.1.1 Cloud providers

At the beginning of the project, a decision was made to allow the project to take advantage of the tools available on the cloud. This was because the nature of the project revolved around users using the system at specific points during the year, where there would be spikes of traffic being directed at the system. By using tools such as automatic load balancing and creating a continuous work-flow through queues, the cloud seemed like an interesting way to take the project.

There were 3 providers that were taken into consideration:

- Windows Azure
- Heroku
- Amazon Web Services (AWS)

1.1.1.1 Windows Azure

Past experience with Windows Azure favoured the use of the system, with the possibility of creating a C# .NET system. However, with no way to have a trial version or free features to test, this did not seem like a safe choice in comparison to the other options.

1.1.1.2 Heroku

Heroku [2] has a free version of their Dynos, with many available add-ons to use. To create the application, a group of Dynos would create a Dyno formation to be used. A Dyno formation for this project would consist of a Web Dyno, and several Worker Dynos. The Worker Dynos would be used to host the queue that is required for the system, as well as the worker itself. Databases are hosted seperately from Dynos in Heroku, but are also free to host. During testing the setup of Heroku, it was found to be difficult to set up the system as desired, with the dashboard not being very intuitive. Furthermore, queues are not a standalone service provided by Heroku, but something that was to be set up within a Dyno, creating complications in how to set up the Dyno specifically for queues.

1.1.1.3 Amazon Web Services (AWS)

Amazon Web Services (AWS) offers a 12 month free trial [3] of their services, which takes into consideration how many hours of computing time is used, along with many other generous limitations. Usage of their Simple Storage Service (S3) [4] is free up to 5GB of storage and thousands of requests, and the Simple Queue Service (SQS) [5] provides a million requests a month. The developer website has many tutorial blogs on how to use certain features, and with tools such as SQS and S3, testing of the system was very successful. For this reason it was the cloud provider that was chosen for the Code Collusion Checker project.

1.1.2 Programming languages and tools

For the Code Collusion Project the following areas needed to be researched, and a decision made on what tools and languages would be used to implement them:

- Website
- Worker
- Database and File Storage
- Version Control

1.1.2.1 Website

As the website was not the major focus on the project, a capable MVC website framework that was easy to set up was required. With experiences in both Ruby on Rails [6] and the .NET framework, these were the two options.

With a years experience creating a C# .NET program, using the .NET Framework would have been a sensible option. However, as discussed in Section 1.1.1, without deploying on Windows Azure, a C# based project did not seem wise. Furthermore, the experience held with Ruby on Rails was much more recent, after completing a module using the framework. Therefore, Ruby on Rails was decided to be the framework to use to create an MVC website.

To develop the website itself, not many tools would be required, as it would be developed on a Linux machine. Once Ruby and Rails 5 was installed on the machine, all that was required was a text editor. There was an option to use the Ruby IDE (Integrated Development Environment) RubyMine. However, there was not much familiarity with this, so the familiar text editor Sublime Text [7] was used.

1.1.2.2 Worker

Similarly to the website, there were two main options to develop the worker: C# and Java. Strengths were similar between C# and Java, however with the decision being made to use Ruby on Rails over the .NET Framework, there was not much backing to using C#. The main advantage to using C# in a system such as this would be the compatibility with the .NET Framework, however this was not required, so Java was the language of choice.

The only tool required for development with Java would be the Eclipse IDE [8]. The main reason for this was the familiarity with the tool as well as the advantages it gives you, such as an integrated debugger and inbuilt running of the code and tests.

1.1.2.3 Database and file storage

Experiences with databases in systems evolve from both SQL and NoSQL databases. However, with the project not being extreme on the amount of data being stored in the database a NoSQL database seemed to be overkill for the system. From the decision that SQL was required, the decision was made based on what Amazon Web Services (AWS) offered as part of their Relational Database Service (RDS) [9]. The main SQL database engine available on RDS was Amazon Aurora, which is Amazon's own SQL engine, which is compatible with MySQL. However, as the project is being developed around the AWS free-tier, the SQL engine chosen was the MySQL database, due to no free-tier option for the Aurora database engine. This is in order to allow for future migration to the Aurora database if it was desired

File Storage was also required in the system, as submissions of code would not be stored in the database, due to the potential size of projects that are submitted. With the option to use AWS already made, the best option available that is provided by Amazon is the Simple Storage Service (S3). S3 is accessible from multiple platforms, with a simple Application Programming Interface (API) to use, which quick research confirmed that it was available for both Java and Ruby.

1.1.2.4 Version Control

The main priority for having version control in a single person project was for backing up the software. Version control itself was not such an important requirement, as there was only one developer working on the code. GitHub [10] was the version control provider of choice for the project, as it is widely used throughout software development, and personal experience has been had in previous projects.

1.1.3 Stanford University's Measure of Software Similarity

While researching the area of plagiarism detection, the only stand-out piece of software that was found was Stanford University's Measure of Software Similarity (MOSS) [11]. Written and maintained by Alex Aiken, MOSS can check for software similarity in a large number of languages via a script written in Perl. The results are then provided via a website link with a comparison of the files submitted, highlighting areas of code that are similar, and can automatically detect code that it expects have been supplied or provided by libraries. This link expires 14 days after creation.

The disadvantage of this system is the accessibility of how the system is used through the submission script. The script itself must be run on a Linux machine, or through Cygwin on Windows, and is not the most intuitive of ways to use the system. Testing through use of this script can be seen in Section 3.3.3 where results of MOSS are compared to the results calculated in the Code Collusion Checker.

1.2 Process

During my initial research for the project, I researched a couple of different methodologies for my project. The process used for the Code Collusion Checker project related closely to Feature Driven Development (FDD). It took the concept of up front design and iterative development to inspire a new version of FDD suitable for a single person project. A major aspect of FDD that has been removed for the single person project was the concept of having roles within a team, and having feature teams. Roles were not required in the project as product and code ownership would be held by a single developer, and this developer would be responsible for every feature that is developed in the project.

By adapting the 5 steps of FDD, I was able to give myself an organised workflow for the project:

Step 1 of Feature Driven Development is to create an Overall Model of the software. This model creates the basis of the upfront planning for the project. From here, every step of planning develops on this model, whether it be a new diagram to further describe the project, or updating diagrams to give more detail about the software.

Step 2 is to create a Feature List of all the requirements for the system. In some projects deploying an FDD approach, the Feature List that is created is final. However, as not all the requirements for the Code Collusion Checker project were fully fleshed out from the beginning, the Feature List was built upon throughout the project. Most of the features found in the list were created during Step 2 of the process, but some areas of the system warranted their own feature later in the project, so the Feature List was added upon.

Step 3 in standard FDD is for feature teams to be created, and for those teams to plan for the features they are to develop. Planning normally involves working within a feature team to delegate class ownership to members within the team, and to have an overall plan on how many iterations might be required to develop the features assigned to them. However, as an Overall Model was created in Step 1, and feature teams were not part of the process created for this project, Step 3 was not required.

Steps 4 and 5 in FDD create the iterative development that the methodology encourages during development. Step 4 is to design the feature that will be implemented, and Step 5 is to develop the

feature. To design the feature, the Overall Model is taken and updated to represent what is planned to be developed in the iteration. For standard FDD, developing in Step 5 involves both developing the required code, as well as testing. However, a lot of the testing for the Code Collusion Project was left until the end portion of the project, as unnecessary or redundant tests were not wanted in the project due to time constraints.

1.3 Analysis

1.3.1 Cloud Computing

From analysing the issues of the problem, the decision was made that the main task in the project would be to ensure that the system is available from any platform, and that the cost of processing the data to check for code collusion was not affecting the system itself.

Having a system that could run continuously without affecting the front-end of the application was extremely important, which is why the decision to run the worker separate from the website was made. By separating the main processing of the application in it's own instance, it allows for maintenance of the back-end while submissions can still be submitted to the website. Furthermore, it synchronises well with an implementation taking advantage of tools available from the cloud.

Hosting the application on the cloud allows for load balancing of the worker, so that when the application is under heavy loads, the worker can be scaled up and out. Scaling up allows for more powerful machines to be used to run the system. Scaling out would allow for more machines to run the same code, to run it more efficiently. The main issue with scaling up and out is having to spend more money on the machines that are being scaled. However, the idea behind scaling is to run less powerful machines until the performance is required. The Code Collusion Checker has been built to suit scaling machines, as the worker would be the only area of the application that would require any kind of scaling. While scaling up machines is easier, scaling out normally provides more efficiency. In this system, by scaling out, more submissions can be checked at the same time, completing an assignment's submissions faster.

1.3.2 Security

Security was the other concern for the application, to ensure that all the code submitted for checking in the system was protected. With the main point of entry into the system being the website, protection was required against SQL Injection, and Cross Site Scripting (XSS). Protection was also required to ensure that free access was not allowed to the database, or any machine running the worker code.

Amazon Web Services provides a security group system, where groups can be given access to other security groups, and restrict access from external access. The main advantage of using security groups is that they are automatically created when a service is created. For example, when a database is hosted a security group is created for database access. When the website is created, another security group is created for website access. To then give the website database access, an inbound rule must be created, specifying that the website is allowed SQL access. By having an easy to use dashboard to manage security within the application helped ensure that the system was secure from any external application that was unwanted.

SQL Injection aims to manipulate parameters entered into a database query to gain access to data on the database, or authorisation to the website. Ruby on Rails has measures to counter SQL Injection [12] that escapes special SQL characters. By using the ActiveRecord Object-Relational Mapping (ORM) framework, and passing an array of strings into the parameters of the ActiveRecord function with the use of the '?' character, the strings are sanitised before creating the SQL command.

XSS attacks aim to inject some code, normally JavaScript, to attempt to steal information from the client, such as the cookie. From this cookie, the attacker can hijack the session by redirecting the client to a fake website, before manipulating other security issues not related to the initial website further attack the client without their knowledge. To prevent this form of attack, Ruby on Rails adopts a policy to use a whitelist of inputs, rather than using a blacklist [13]. This ensures that you are only accepting expected inputs from the website, and filtering out any unknown parameters.

1.3.3 Feature List

Once taking all these areas into consideration, a Feature List was created. Initially, there were a total of 10 high level features, with more being created throughout the project. The features that were created did not go into too much technical detail about what would be required to complete the feature, so that it was readable by a wider range of people who might not have such a technical background.

Within the feature list, a priority and complexity rating, as well as a note as to what iteration the feature was implemented. Notes were given where necessary for a feature, and all features had a summary with them.

Code Collusion Checker - Feature List							
Feature No	**Feature Name**	**Summary**	**Priority**	**Complexity (1-5)**	**Iteration**	**Completion**	**Extra details**
1	User Accounts	A user needs to be able to hold an account on the system, and be able to sign in using this	Med	3	4	Completed	
2	Creating assignments	A user needs to be able to create assignments in the system. An assignment will have an associated programming language. Submissions must be uploaded against an assignment.	High	1	2	Completed	
3	Uploading and using templates	Assignments need to be able to hold information about whether a template was provided. This template will need to be stored in S3, and have a reference to it in the assignment object.	Med	3	6	Completed	
4	Uploading single submissions	A user should be able to upload a single submission to an assignment.	High	2	1	Completed	
5	Uploading multiple submissions	The system should be able to handle more complex uploads. This includes but is not limited to a zip file containing zip files associated with submissions	Med	3	5	Completed	
6	Checking submissions via string matching	Submissions need to able to be checked using string matching against other submissions in the assignment.	High	4	3	Completed	An implementation of the Levenshtein Distance has been developed, and feature #14 works towards improving this
7	Checking submissions via code parsing	Submissions should be checked against parsing methods	Med	5	7	Not started	Research has been completed into how this might be implemented
8	Reviewing results of an assignment	Once code collusion checking is complete, results should be viewable on the website.	Med	2	2	Completed	A basic results view has been implemented, and feature #13 has been created to develop this further
9	Exporting results of an assignment	Results should have the capability to be exported, to multiple formats (xls, csv etc)	Low	2	Unassigned	Not started	
10	Uploading submissions with multiple files	Submissions need to be able to be uploaded with multiple files, representing a project	High	4	6	Completed	
11	Retrospective collusion checking	During collusion checking, if a percentage against a student is found to be higher than previously checked, this should be updated	High	3	7	Completed	
12	Worker Configuration File	The backend worker should have a configuration file to input database and AWS details	Med	2	5	Completed	
13	Improved results view	A more in depth results view should be created to make the website more usable	Low	2	6	Not started	
14	Enhancement of String matching	With a complexity of O(n!), the string matching algorithm needs enhancing to improve speed efficiency	Med	5	5	Completed	An attempt to enhance the algorithm was completed but was unsuccessful at improving upon the original algorithm used

Figure 1.1: The feature list created as part of the FDD-like methodology

Chapter 2

Design & Implementation

Due to following a methodology similar to Feature Driven Development which was discussed in section 1.2, the design and implementation chapters of the report have been merged. For each iteration within the project, a section has been created to report on the progress made in the project. For each iteration, a design phase and implementation of features phase occurred, and sections have been created to reflect this.

2.1 Iteration 0

As part of following a Feature Driven Development (FDD) approach, the first iteration of the project was to create an overall model of the application, that could be developed and improved throughout each following iteration. Once this was completed, the tools to be used throughout the project would be set up, so the environment would be ready for development in Iteration 1.

2.1.1 Initial design

The initial design began with a simple use case diagram, describing how the lecturers will be the only users in the system. There are only four use cases that a lecturer has for the system: login, create assignments, uploading code, and reviewing results. The only other system level diagram created in the initial design phase is a sequence diagram, which goes into more depth for the use cases as to how the application will run in the required situations. These two diagrams describe the work-flow of the system as a whole, to ensure that as the complexity of the system might progress, there are diagrams that can be referenced to make sure the requirements of the system are not altered.

The next stage was to design the front end, back end, and database for the system. Rather than creating detailed UML diagrams for these, an agile modelling approach was taken. This idea taken from John Hunt's Agile Software Construction [14] takes the approach that an initial design is created in it's basic form to then be developed further when a better understanding of what is required is known. This allows for adaptability of the structure of the software, and fits extremely well in the FDD approach being taken.

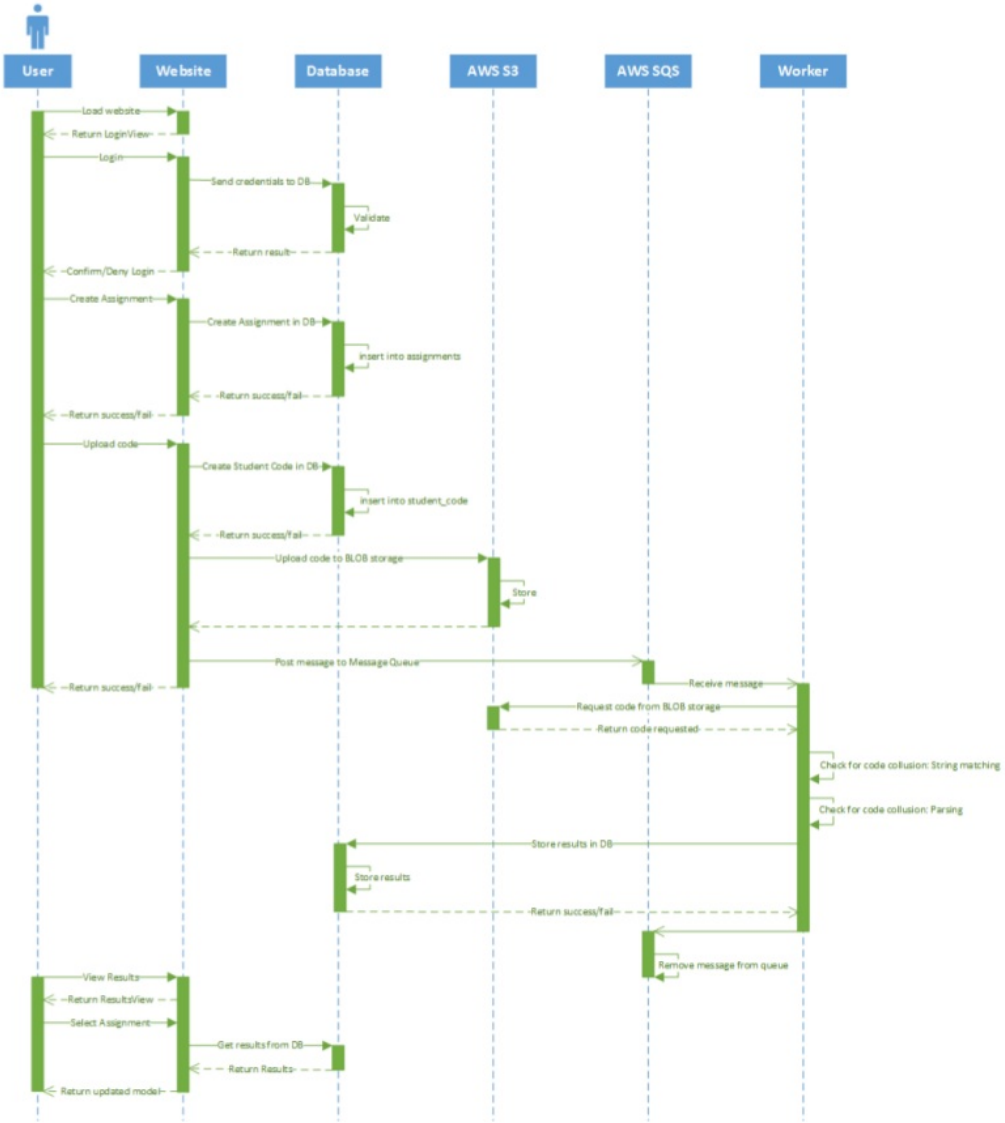

Figure 2.1: The system-level sequence diagram created as part of the overall model

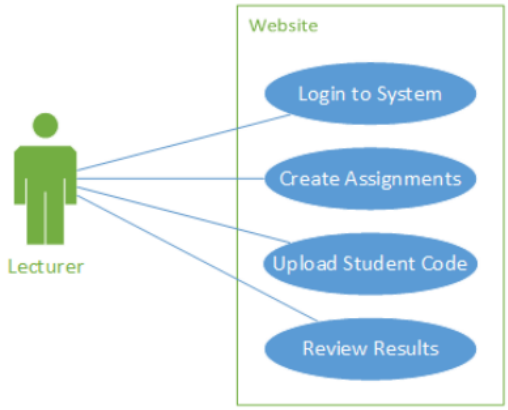

Figure 2.2: The system-level use case diagram created as part of the overall model

2.1.1.1 Database

The entity relationship diagram initially given for the MySQL database simply provided the main database table names. These tables would then be mirrored onto the website, where the data could be managed. However, to begin the project, having users in the system would not be a priority, so although they have been included in the overall model from the beginning, they would not be implemented until later.

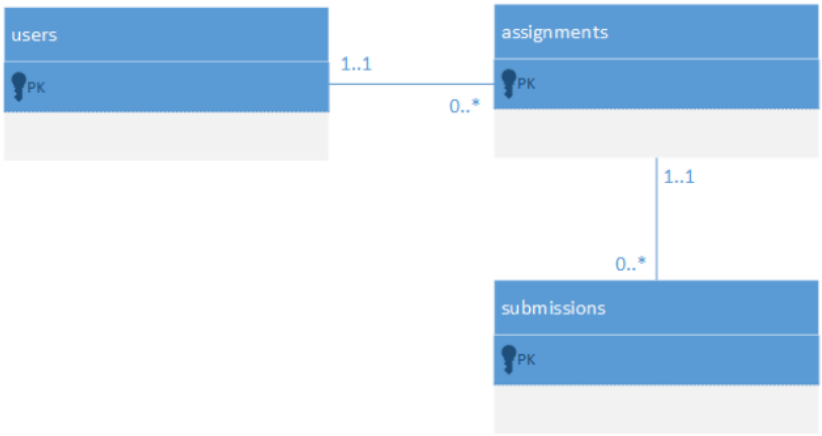

Figure 2.3: The initial entity relationship diagram for the database

2.1.1.2 Website

The MVC website has a UML class diagram to represent the controllers, views, models. The class diagram for the website for the overall model acts more as a representation of the architecture of the website, instead of providing details about what code some of the models and controllers would be executing. As the website was to be the point of access for the application, it was preferential that it wasn't complex. By not overcomplicating the design of the website, this allows for the correct structure to be implemented, while being flexible to change where required.

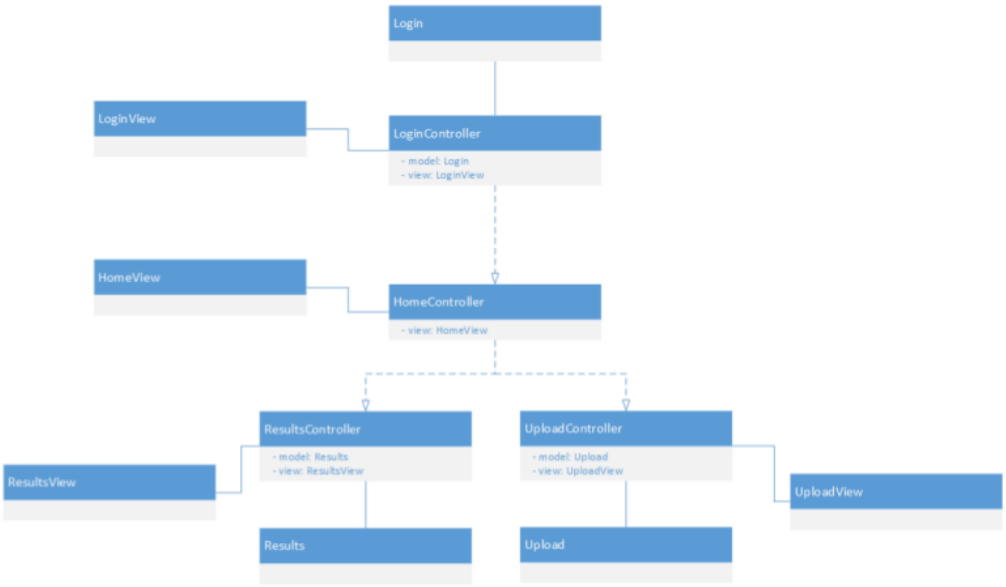

Figure 2.4: The initial class diagram for the MVC website

2.1.1.3 Worker

The Java back end also has a UML class diagram, created to give an initial idea as to what classes would be required for the system. The initial concept brought into this design at the early stages was to develop a system where both maintenance of the system would be easy, and expansion of what would be processable. For example, by providing an interface of string and parse checkers, but only implementing a Java implementation of these checkers the system has the opportunity of expansion in the future to be able to cover other languages.

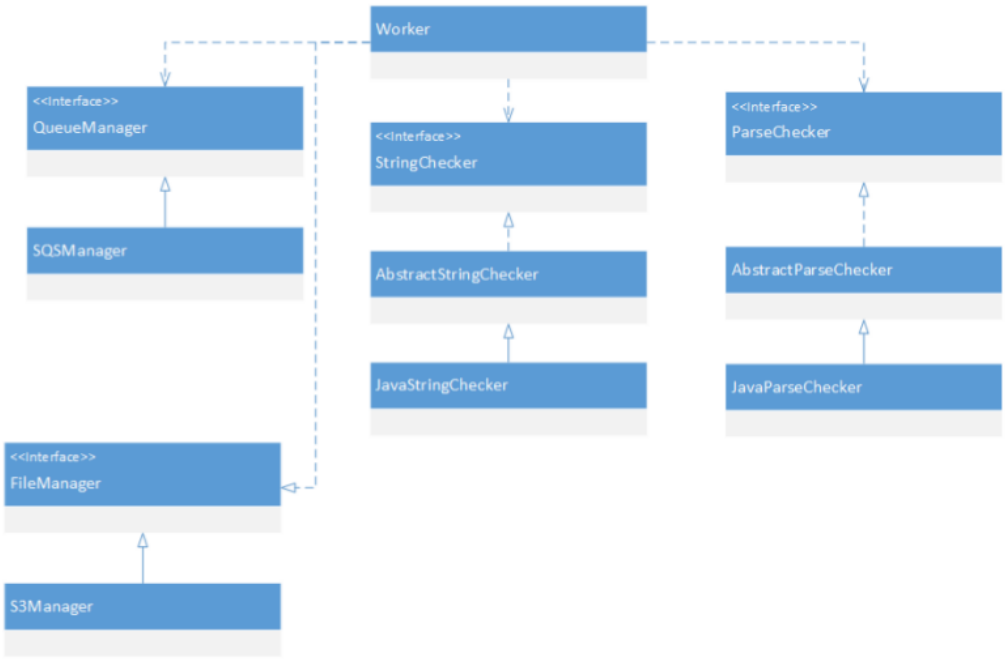

Figure 2.5: The initial class diagram for the Java back-end worker

2.1.2 Initial setup

Once the overall model was created, time was spent setting up the environment that would be used for the majority of the development phase. The tools set up in the environment at the end of Iteration 0 were:

- GitHub
- Eclipse
- Ruby on Rails

2.1.2.1 GitHub

Using the GitHub website, a private repository was set up for the project. This repository was then given 2 branches, the master branch and a development branch. During an iteration work would be completed on the development branch, and at the end of every iteration a pull request would be made to merge development back into master. This pull request would then act as a summary of the iteration, describing what had been achieved, and what would be required in the next iteration. The pull request would then be merged into master, and the new iteration would begin again on the development branch.

2.1.2.2 Eclipse

As discussed in section 1.1.2.2, Eclipse was to be used throughout the project to develop the Java back-end worker. By using the Eclipse IDE, creating and managing Java projects would be a lot easier, as well as managing the version of Java the project would be running on.

2.1.2.3 Ruby on Rails

To develop the website, Ruby on Rails was installed on the development environment. Ruby version 2.3.1p112 and Rails 5.0.1 were installed, in preparation for developing the website. Sublime Text was also installed as the text editor of choice for creating the website.

2.2 Iteration 1

Iteration 1 was the first iteration that implemented code. The main goal of the iteration was to create a code base to work from, and through spike work, create and test a work-flow of the system that could be used as the base of the project.

2.2.1 Upfront design

Due to the nature of the early iterations, the upfront design for Iteration 1 can be taken directly from Iteration 0. By not updating the design, the developing of code was much more flexible for the spike work, so that while the initial design was being adhered to, there were no strict regulations as to how the code might look.

2.2.2 Feature #4: Uploading single submissions

A user should be able to upload a single submission to an assignment

To complete Feature #4, the initial work-flow of the system needed to be completed. Iteration 1 built this work-flow, which consisted of:

- Website
- Database
- Message queue
- BLOB storage
- Worker

2.2.2.1 Website

The website was created as an MVC website using Ruby on Rails. Due to recent experiences with Ruby on Rails, the implementation of the website was not complex, and was designed to be simple, and easy to develop.

As the project was to be developed on Amazon Web Services (AWS), there were two options for website hosting: Elastic Beanstalk (EB) [15] or Elastic Compute Cloud (EC2). After looking at both options on AWS, EB was chosen to host the website as EB uses EC2 instances to host a website for you, while providing a much easier configuration. The guides provided by AWS for EB [16] were used during this spike work, providing guides on how to set up a command line interface on a development machine to easily update the website at any time, and how to set up the Ruby on Rails website to be compatible with EB.

2.2.2.2 Database

As discussed in Iteration 0, an SQL database had been chosen to be the optimal decision for the project. AWS offers two options for a relational database: Aurora or Relational Database Service

(RDS). The decision was taken to use RDS because there was a free-tier option, and RDS provides the option of using the MySQL engine which was familiar to use.

AWS also offer security groups within the application to restrict access to certain areas of your system. A security group was set up for the RDS instance, where access to database was restricted so only the website, worker, and developer machines could access the database.

Connecting to the database from the website was completed through the configuration files in the website, specifically database.yml. For Java, the Java Database Connectivity driver for MySQL [17] was used.

2.2.2.3 Message queue

The message queue is required in the system to send messages between the website and the worker, so that the worker knows information about submissions it will be checking.

One of the primary reasons for choosing AWS as the cloud host to use was for the usability of the Simple Queue Service (SQS) they offer. This was set up within the AWS website, where a queue was created with restrictions that only machines that provide credentials for the main account can access the queue. For the website to connect to the SQS queue, the AWS SDK for Ruby [18] was installed as a gem. From here, a guide [19] was followed to help shape how to send messages to the queue, for the Java worker to receive. The worker used the AWS SDK for Java [20] to provide functionality capable of receiving these messages. Amazon provided a guide [21] to aid writing the communication code required to do this.

2.2.2.4 BLOB storage

Binary Large Object (BLOB) storage is required in the application to store the submissions that will checked by the system. The submissions will be uploaded via the website, and then downloaded again by the worker.

AWS offers their Simple Storage Service (S3) to be used for exactly this. Much in the same way that SQS works, only machines that provide the required credentials will be granted access to the file stored here. Using the AWS SDK for Ruby, the website could upload files to S3, as shown in Amazon's guide to uploading files to S3 in Ruby [22]. The Java worker was only required to download files from S3, and following Amazon's guide [23] regarding this, the AWS SDK for Java was used to accomplish this.

2.2.2.5 Worker

The worker was created to be a separate entity from the website, so that checking of submissions could be completed without affecting the performance of the website. Feature #4 requires the worker to be capable of reading the submission once it has been uploaded.

Created as a Java project, the plan was to upload the code to an EC2 [24] instance, that would run the code in a constant loop. However, for Iteration 1 this was not required or possible. In order to get the worker in a state where constant reading of code was usable, a better idea of what will be received from the website and a more established database structure was required. Therefore

for the initial version of the worker, it was only run locally to test whether the submission was able to be received.

2.2.3 Retrospective

Looking back on the iteration, the main challenges that were faced in the spike work were focused around learning AWS, and details about how they worked within Ruby and Java. The documentation blog posts provided by Amazon helped get over these difficulties, and the spike work created a strong code base to work from.

2.3 Iteration 2

Iteration 2 focused on the website, so that the worker could be developed within Iteration 3. The development on the website aimed to give the website a more user-friendly feel, through the use of Bootstrap. The infrastructure of the website would then be expanded to create pages for assignments and submissions separately, so the worker would receive more realistic data to be developed upon.

2.3.1 Upfront design

In Iteration 2, the website was the main focus, but this also impacted the database. The UML Notation for the database was updated to describe the columns required for both the assignments and submissions tables. This included the foreign key to create the relation between the tables. The class diagram for the website was also updated to introduce the submission and assignment controllers. The model was also updated with required method names.

Figure 2.6: Iteration 2 Entity-Relationship diagram introducing columns to assignments and submissions

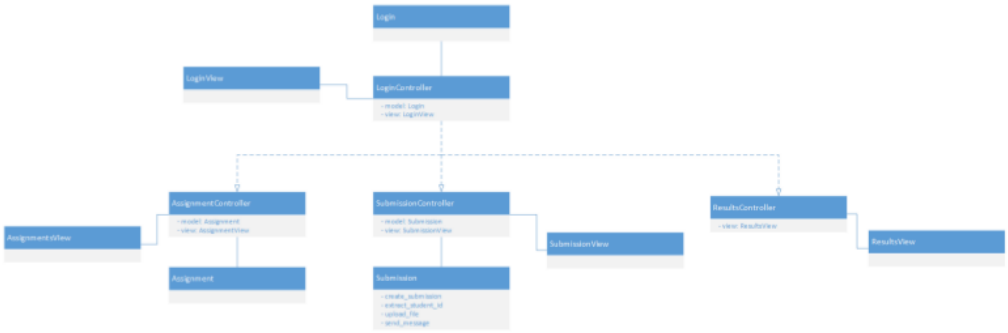

Figure 2.7: Iteration 2 website class diagram adding new controllers, and Submission model methods

2.3.2 Feature #2: Creating assignments

A user needs to be able to create assignments in the system. An assignment will have an associated programming language. Submissions must be uploaded against an assignment.

The first step in the iteration was to add the CSS and JavaScript files for Bootstrap [25]. This made the creation of web pages a lot easier, and allowed for the creation of a simple navigation bar for the site.

Once Bootstrap was implemented, scaffolds were generated using Rails to create the basic structure required for assignments and submissions. After fine tuning what was required through the use of migration files, the website was capable of creating assignments, which submissions could then be created against.

2.3.3 Feature #8: Reviewing results of an assignment

Once code collusion checking is complete, results should be viewable on the website.

While updating the website to a usable state, a results section was created on the website and database, ready to receive results about a submission that has been checked. The results do not have their own table in the database, but instead a submission has collusion_percent and collusion_details columns to allow for result details to be held against a submission directly.

This results page is only a basic version of what is required by the end of the project, as multiple results will be available for multiple users. Feature #13 has been created to ensure that the results page is updated when more complex results are created by the worker.

2.3.4 Retrospective

Using migration files to get the database to a desired format took a little longer than expected with Ruby on Rails. This was not so much to do with the syntax in how migrations worked, but more about how they should be used. There was an issue during the iteration where the database would not migrate while trying to delete a table that didn't exist. It was being deleted as it was a table that was no longer required once the format of the database had been decided. However, the table only existed on the current machine after the table had been created manually onto the MySQL server. Therefore, when the website was hosted on AWS, there was a mismatch in database's, and the website would not run. Once the issue was understandable, the simple fix to the problem was to delete the migration file that was causing the issue.

Apart from this issue, the iteration was successful, allowing for further development to start on the worker.

2.4 Iteration 3

The objective of Iteration 3 was to get the system to a point ready to be presented in the mid-project demonstration. This would require string matching to be completed, and for the code to be run on AWS.

2.4.1 Upfront design

The class diagram for the worker was vastly expanded on at the start of Iteration 3, as it was now known what information was being given to the worker to work with. After researching different design patterns [26], factory classes and interfaces were chosen to be used through the application for the 4 main areas:

- Database
- File
- String Checker
- Parse Checker

The reason behind choosing factories was that it gave the application the flexibility and capability of expanding further in every direction. It allowed for developers to update the software at a later date to include a different kind of database, file manager, or to be able to check more languages for both the string checker and parse checker.

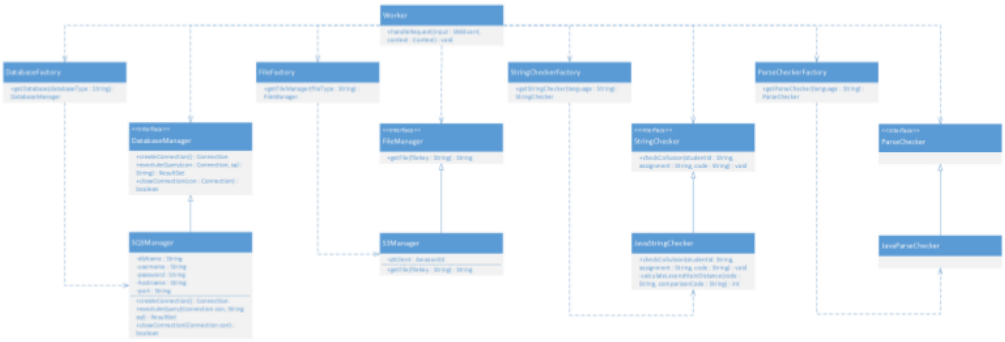

Figure 2.8: Iteration 3 worker class diagram introducing factories and interfaces with methods

2.4.2 Feature #6: Checking submissions via string matching

Submissions need to able to be checked using string matching against other submissions in the assignment.

After implementing the structure discussed above, the main task within the iteration was to implement a form of string matching. Research around the area pointed towards creating an implementation of the Levenshtein Distance [27].

The Levenshtein Distance is a measure of how similar two strings are, by calculating an edit distance between them. The edit distance is the number of additions, deletions, and edits required to make one string identical to the other. This can be seen in the below matrix, where the value found in the bottom most right cell is the final edit distance.

		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

Figure 2.9: Matrix showing path of Levenshtein Distance [1]

The matrix defines a path through the changes of the strings, where a cost is defined for every move. These movements are defined as:

- A diagonal movement describes either a change of character, or no change at all
  - A change costs 1, no change costs 0
- A horizontal movement describes an insertion of a character to the original string
- A vertical movement describes a deletion of a character from the original string

To ensure that the Levenshtein Distance was optimal for the system, the version developed in Iteration 3 was based from Michael Gilleland's essay and example source code [28]. This implementation creates a literal 2D array of integer values, which represents the matrix seen above. From here, the the matrix is completed via a brute force method, by calculating every value in the matrix, to be capable of finding the final distance.

2.4.3 AWS Lambda

Once Feature #6 was completed, the code was ready to be uploaded to Amazon Web Service (AWS). Originally, the plan was to upload the code to an Elastic Compute Cloud (EC2) machine to be run from the command line. However, after looking into other services provided, AWS Lambda [29] looked to be a more efficient solution. Lambda runs code on demand, and does not require the infinite loop that a Java program would require for this kind of application.

A Lambda project runs on demand by using a trigger. When the trigger is updated, the code is run against information given by the trigger. Ideally, this trigger would have been from the Simple Queue Service (SQS), but this is not offered by AWS Lambda. Therefore the project was updated to run against the Simple Notification Service (SNS) [30] that AWS provide. SNS runs very similarly to SQS, as notifications are posted to the service in an almost identical way to the queue service. SNS differs to SQS because the notifications are pushed to clients, rather than clients polling the queue.

Lambda also requires it's own kind of project, available from the AWS Toolkit for Eclipse [31]. This project runs from a handleRequest method, taking parameters relating to the trigger. For this system the trigger is SNS, and the message within the notification is read. This message contains the ID of which submission requires checking.

2.4.4 Retrospective

Iteration 3 completed the entire work-flow of the application, creating a runnable application that can check code on a basic level, using the Levenshtein Distance. However, using a brute force method to complete a 2D array might be taxing on the system, dependant on how large the strings to compare are. Therefore during Iteration 4, an implementation of the Levenshtein Distance will be developed, where the optimal path is found while not having to search the entire matrix. This should reduce the time taken for a comparison. Feature #14 has been created to mark that this is required.

Another decision that was required was to also refactor some of the factory classes that have been created. There are some manager classes such as DatabaseManager and FileManager which will be better represented as singleton classes.

2.5 Iteration 4

Iteration 4 focused on refactoring of the worker to more efficient data structures. It also started development on a new version of the Levenshtein distance. Additionally the website was updated to introduce the concept of users into the system, who would have their own assignments to look at

2.5.1 Upfront design

The back end had a lot of refactoring occur, which came about with a large update of the class diagram. This refactoring aimed to create more maintainable and understandable code, for when more complex features would be added into the system. The notable refactors were the introduction of singletons for both the DatabaseManager and the FileManager, and the implementation of a Submission class, to remove a lot of the ArrayLists of Strings, so they were containable within a class.

The database and front end were both updated to accommodate for users being brought into the system. The website has a sessions controller as well as the users controller, where the session manages the logging in of users, and the users controller manages the creation of users.

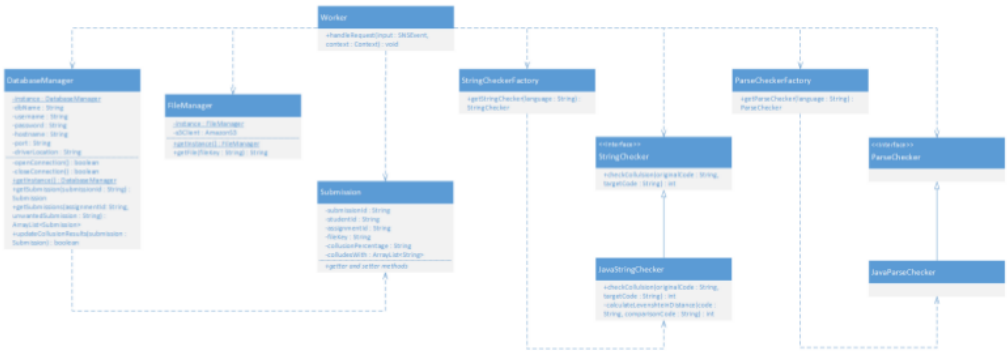

Figure 2.10: Iteration 4 worker class diagram introducing singleton managers

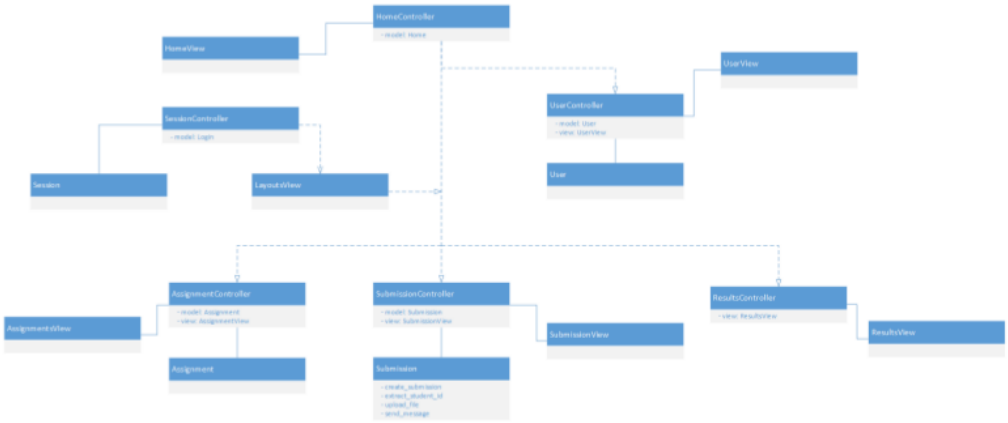

Figure 2.11: Iteration 4 website class diagram introducing users and sessions

Figure 2.12: Iteration 4 database entity-relationship diagram updating the users table

2.5.2 Feature #1: User Accounts

A user needs to be able to hold an account on the system, and be able to sign in using this

In order to make sure that users were not seeing assignments that might have been created by another user, which was irrelevant to them, it was essential for users to be a part of the system. This also protects the data that is being transferred and used by the system.

Users were created with the aid of the bcrypt gem [32], to create secure passwords, using the ActiveModel method has_secure_password. Providing that the user model in the database contains the field password_digest, the has_secure_password method within bcrypt then encrypts the password through it's hashing algorithm. A tutorial written by Greg Buckner [33] was followed to then understand how users could be implemented into a Ruby on Rails system with the use of the bcrypt gem.

Sessions for the users were created alongside the users, so that once the users were created, sessions could manage the authentication and authorisation of a user. Authorisation is completed via the ApplicationController, using a before_filter method on all controllers that require authorisation. The before_filter points to an authorise method on the ApplicationController, which checks to see if there is a session currently logged in. If there is, the user is authorised to access the page. Otherwise the user is redirected to the login page, where they can login. This login checks the

user's authentication details to see whether they should be logged on. These details are the email address and password.

2.5.3 Retrospective

The refactoring done within Iteration 4 was an important part of the project, as it make the Java code a lot more manageable, and more object-orientated. However, the refactoring did take longer than expected along with the introduction of users, so development on the Levenshtein Distance was not possible. This has now been moved to Iteration 5 as a priority to be completed.

The users being added into the system was a vital from a security perspective, so there was a form of authentication and authorisation in the system. Getting the sessions to work as desired was a challenge, as there was no time-out on the session to begin with. Once the cookie was located, a time-out was introduced and the sessions with users worked as intended.

2.6 Iteration 5

Iteration 5 introduced a configuration file for the worker, that contained details for connecting to the database, and AWS Simple Storage Service (S3). Most of the iteration was spent designing, implementing, and testing a different algorithm of the Levenshtein Distance.

2.6.1 Upfront Design

The design did not change significantly for the iteration as the new algorithm for the Levenshtein Distance would replace the current version, not affecting the structure of the code itself. To use the configuration file, a ConfigurationManager class was created as a singleton.

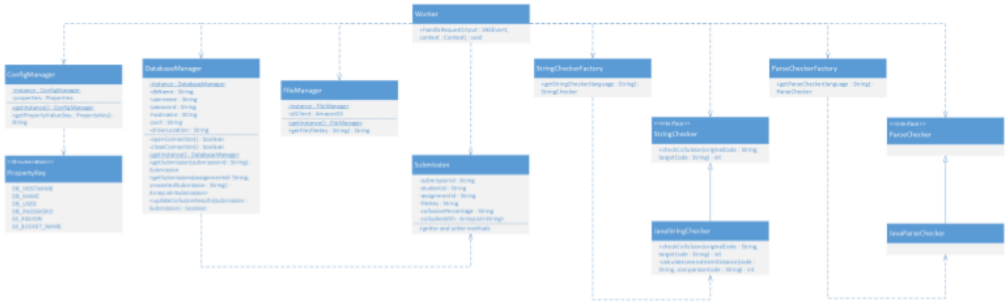

Figure 2.13: Iteration 5 worker class diagram adding the ConfigurationManager

2.6.2 Feature #12: Worker Configuration File

The backend worker should have a configuration file to input database and AWS details.

The configuration file was created as a .properties file, and loaded into a java.util.Properties variable within the ConfigurationManager class. An enum was then created of every property key within the configuration file. The ConfigurationManager was then the only point of access to the properties file, and other parts of the system would call the singleton class when it required any of the properties. This ensured that any file reading that was happening in relation to the configuration files was kept within the class.

The system previously was configured using environment variables, which is the same method used for the website. Therefore the decision was made that the if the configuration file is not available, then environment variables would also be checked. This acts as a backup for any situation where the configuration file is not available.

2.6.3 Feature #14: Enhancement of String matching

With a complexity of O(n!), the string matching algorithm needs enhancing to improve speed efficiency

As the time complexity of n submissions in an assignment is O(n!), it was worth spending the majority of an iteration on an attempt of an improved algorithm of the Levenshtein Distance. This

code can be seen in Appendix C Section 3.1.

The concept behind the new algorithm was to still create the 2D matrix and to start at the beginning of the matrix, at the coordinates (0,0). However, rather than brute forcing through the matrix to find the last value in the matrix, the path of least resistance. For the Levenshtein Distance, this means that any value in the matrix which does not cost anything will be valued more.

Once the algorithm had been developed, it was found through testing that it was slower than the original brute force implementation. Through analysis of the code, although less values are being calculated in the string comparison itself, the calculations to work out what values should be being checked are costing more than the brute force method of checking every value.

2.6.4 Retrospective

Iteration 5 did not introduce any new features to the system that an end user would notice, so overall the iteration did not feel like a very successful one. However, the introduction of configuration files cemented the maintainability of the code. Furthermore, by developing the new Levenshtein Distance algorithm to attempt to speed up the code gave a better understanding of how the checking itself worked, and ensured that the optimal method of execution was being used.

As the original version of the Levenshtein Distance was found to be faster, it was kept in the code as the version to be used, but the algorithm developed for Feature #14 was kept in the technical hand-in.

2.7 Iteration 6

Iteration 6 planned to tackle the last major technical area before looking at code parsing. This involved updating the website so it is more user friendly from the perspective of uploading files. Before Iteration 6, it was only possible to upload single file submissions, which was due to work being prioritised on the worker. Due to work completed on string matching in Iteration 5, multi-file uploads of multiple file submissions was required, along with file templates.

2.7.1 Upfront design

With changes to how submissions were being uploaded to the system, it was decided that results and submissions should be separate entries in the database and website, with a foreign key in the results table pointing to the related submission.

A migration was made to add a column to the assignments table, called template_key. This column holds the file_key that is created when storing the file in Amazon Web Service's Simple Storage Service (AWS S3).

Structurally, the website and worker were not changed. However, models and methods were included and updated.

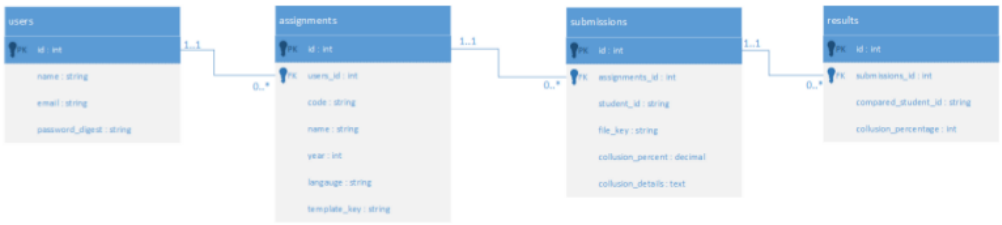

Figure 2.14: Iteration 6 database entity-relationship diagram adding the results table and updating assignments

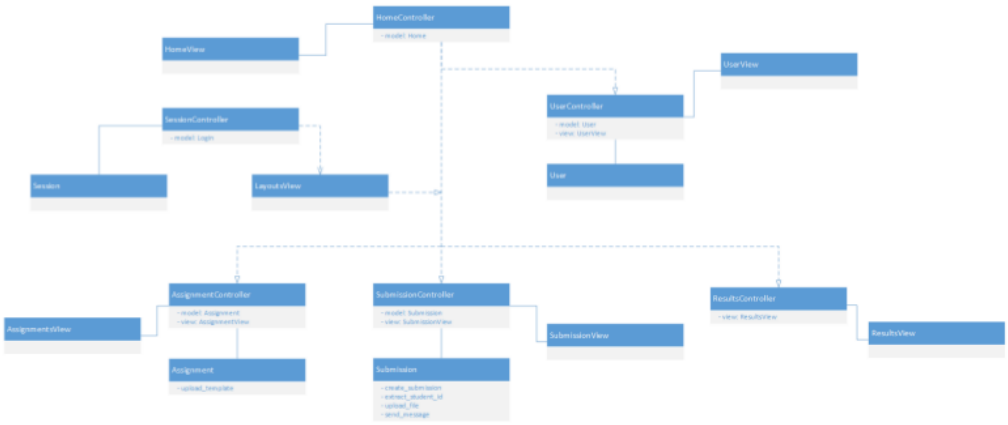

Figure 2.15: Iteration 6 website class diagram adding template method to assignments model

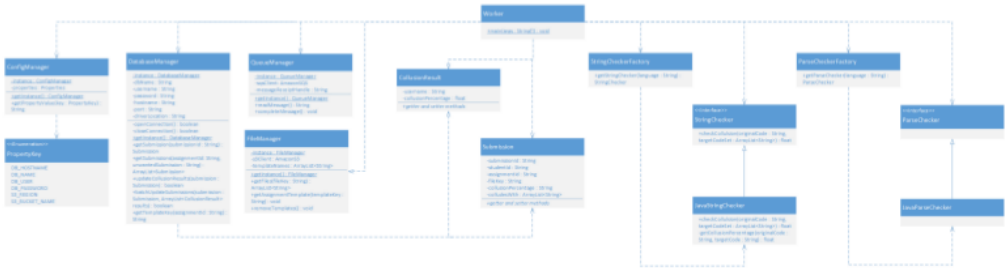

Figure 2.16: Iteration 6 worker class diagram updating to accommodate for templates and multi-file projects

2.7.2 Feature #3: Uploading and using templates

Assignments need to be able to hold information about whether a template was provided. This template will need to be stored in S3, and have a reference to it in the assignment object.

Template uploads were designed to be uploaded as a zip file, and to be stored on S3 for the worker to access. Once updating the form for assignments to allow for file upload, the controller was updated to check if the file upload had a file within the form. If so, a new method created in the assignment model uploads the template via the aws-sdk gem, in the same way that files have been previously uploaded. However, they are uploaded to a different bucket within S3, so they are not located in the same area as submissions.

Creating the assignment became an issue, as the file was not being uploaded to the database. Therefore, it was not part of the expected model. This mean that using the assignment_params variable will not work, as it contains the template file, which it does not know about. Instead, the assignment needed to be created using the variables within the assignment_params.

Once the templates were uploaded, the worker then needed to handle downloading the zip file, and using the template to remove code not written by the student. The process to use the template was written through the following steps:

1. At the start of checking a submission, any template files for the related assignment would be downloaded from the Simple Storage Service (S3) to be used. The downloaded template file names would then be stored in an ArrayList.
2. When checking each file in a submission, if the file name matches any of the template files, then a Linux diff command would be run within the Java code [34].
3. The diff command compares the template file with the submission file, and only takes the additions from the output. The output is then cleaned from the unwanted symbols, and outputted to a new file.
4. This new file is then opened and the string within it is read, creating the comparable string of code that needs to be checked.

Once this had been implemented and uploaded to AWS Lambda, the code failed to run. The reason behind this was that because of the on-demand way that Lambda executes code, there is no local storage for files to be outputted and read from. Therefore the diff command could never

write it's output to a file for further use. This also brought the attention of another issue, where the ConfigurationManager was not able to use the properties file in a way that was required. Instead it was relying on back-up environment variables.

Due to these issues, the decision was made to move the project back to an Elastic Cloud Compute (EC2) instance, which runs as a virtual machine (VM). By re-implementing an infinite loop in the code, a constant work flow was achievable by reading the Simple Queue Service (SQS) that would also be re-implemented.

2.7.3 Feature #5: Uploading multiple submissions

The system should be able to handle more complex uploads. This includes but is not limited to a zip file containing zip files associated with submissions

Being able to upload multiple submissions for one assignment at the same time was vital for the project, to make it a practical tool to use. Although the final code was simple to implement, a couple of issues arose from this.

The first step to was to change the form to accept multiple files. Initially this was done by adding the multiple: true option to the file_field_tag. However, the controller did not read the parameter correctly, as it did not read the parameter as a collection of files. After realising that this was the issue, the field tag was set to be an array of upload.

Once this was achieved, by iterating over every file uploaded, each submission could be uploaded and saved to the database separately.

2.7.4 Feature #10: Uploading submissions with multiple files

Submissions need to be able to be uploaded with multiple files, representing a project

Having submissions with multiple files was also vital for the system, so that it was possible for entire projects to be uploaded for a single project. Furthermore, being able to specify which file types would be read from a project is also important. This will ensure that libraries that might be included in a project's source code would not be checked by the checker, as well as any project files.

Uploading multiple files for a submission on the website does not require any structural changes, as the website does not deal with the files themselves, but instead uploads them to the Simple Storage Service (S3). A hint was updated near the file upload on the form to inform users that they are required to upload submissions in a zip file format.

The significant updates for multiple files occurred in the worker, where zip files would have to be read from S3. As mentioned in Feature #3, local file manipulation was not possible on Lambda, which was required for unzipping files from S3. In order to read a zip file from S3, the header information for all the files was first retrieved using a ZipInputStream. By then iterating over each file's information, it was possible to see if the file extension matched the type that was required. For every file that needs to be checked within the zip, a java.io.File is created within the project, and the file of interest is written to the new file, so it is accessible outside of the zip, as an uncompressed file. Any template is then applied as mentioned in Feature #3, before a StringBuilder creates a string of a file with the aid of a RandomAccessFile object. This code can

be found in Appendix C Section 3.2.

This convoluted method was the only way that zip files were readable from S3, as the S3 object presumes that the objectContent by a FileInputStream, rather than a ZipInputStream. Furthermore, it allows for a filtering of files so that only files that relate to the language of the project are checked.

2.7.5 Research into parse checkers

2.7.6 Retrospective

Iteration 6 brought about the finishing of technical work from a standpoint of functionality, where the project can be seen as a working prototype of the desired product. By implementing the multiple file uploads, and allowing for multiple submissions to be uploaded at once, the system felt very user friendly. Furthermore, from this point, the only part of the website that would need updating is the how the results are shown. However, as this is a low priority and due to time constraints, this did not look likely to be achieved for the project.

2.8 Iteration 7

Iteration 7 did not bring about any new features, and was focused around testing the application. Research was also put into parse testing, to see how viable implementation of a parse checker would be in the time remaining in the project.

2.8.1 Feature #9: Checking submissions via code parsing

Submissions should be checked against parsing methods

As there was not much knowledge in the area of parsing, research was completed on the topic to see what might be suitable for the task, and how it might be accomplished. JavaCC [35] and ANTLR [36] were two libraries that were looked into as to what could be required. While a decision was never made into what library would be used, after looking into how the checking may be done, it was decided that the logic to implement such a checker would require too much time.

If the logic were to be implemented, it would work around the concept of creating a parse tree, that would be created and navigated by the libraries. A comparison of how the code runs regarding method workflow, and return types could then be compared between submissions to see if the logic behind the code is identical. For template driven code however, this approach may not work as the basis of how the code works will have been provided by the lecturer.

2.8.2 Testing

While manual testing of code was completed during each iteration, more automated tests were not completed until Iteration 7 of the project. This decision was made as redundant tests could have been made throughout the project while major refactoring occurred, and when the structure and flow of the project changed regularly. Detail of what testing was implemented in this iteration can be found in Chapter 3.

Chapter 3

Testing

3.1 Overall approach to testing

Within a standard Feature Driven Development (FDD) approach, testing would be done per iteration as part of the implementation of the features. This was achieved to a degree in this project, the approach taken to testing in the project was executed differently. Manual testing was completed at the end of each feature that had been developed, and worked as intended. However, automated testing was not introduced into the system until the later stages of the project

The decision behind automated testing not being created was taken from an early standpoint, when the final structure of the software was not fully known. This decision was taken because it was believed that by writing these tests early in the project with the amount of refactoring that would be required, that a lot of the tests would become redundant code that would not be used. While this was true that a lot of the structure was refactored to the point where a lot of the tests that would have been created would be surplus to requirement, the code base itself would be much more manageable and maintainable for future development.

3.2 Automated testing

Automated Testing was completed in the two testable areas of the project: the back-end worker and the website. The worker was tested through Java's unit testing framework JUnit [37], and the website was tested using Cucumber's automated acceptance tests with the help of Rails' rails-cucumber gem [38].

Due to time constraints and the approach taken to automated tests, there were few automated tests that were created. The tests that have been created were testing the critical workflow components of the system to ensure that there was some form of testing behind them.

3.2.1 Unit tests

While creating JUnit tests for the worker, it was difficult to test some functionality which interacted with Amazon Web Services. Due to the set-up using cloud provider, a whole test environment would be required in order to be able to test some functionality while not affecting the running

version of the software itself. This was an oversight of developing with dependencies within the cloud, as it meant that there was no possible testing available for any functionality of the FileManager which uses the Simple Storage Service (S3), or the QueueManager that uses the Simple Queue Service (SQS).

The other mistake in creating JUnit tests so late in the project was the lack of refactoring that was completed for testing purposes. If the structure of code in the project was refactored, more in depth JUnit testing throughout the worker would be possible. Nonetheless, due to time constraints, and the system working fully functionally, this was not a priority

3.2.2 Cucumber testing

Unit tests were not implemented for the website, as there were not many methods that were created and testable with the issues regarding AWS. Instead, acceptance tests were written through the Cucumber testing framework. Acceptance testing was more important for the website because in the project, the website was the only access point of the software, and using Cucumber testing allowed for expected usage of the website to be tested.

A similar problem to the Unit Tests were found with accessibility to AWS from a testing perspective. Ruby on Rails allowed for a test database to be created and used, so creation of users and assignments was possible. However, the core process behind creating submissions was to upload the submission itself to AWS S3. As explained in section 3.2.1, without a test environment set up on AWS, testing of this area of the website was not achievable for the project.

To write the Cucumber tests themselves, feature files were created for every feature in the website that was considered to require testing. As the feature to create submissions was not possible to code, a feature file for submission creation was made to be completed in the future. Step definitions were then created to test the features described in the feature files.

3.3 Manual testing

Manual testing was completed during every iteration as part of development, ensuring that features that were being completed as part of FDD were behaving in the desired manner. Leading on from this, a stress test was completed to see how the system would cope with a large workload, and the Measure of Software Similarity (MOSS) system was tested and results were compared between the two systems.

3.3.1 Iterative

During implementation of every iteration, there was constant testing done manually to ensure that the code which was being written was acting in an expected manner. This included debugging code that was written using the Eclipse debugger to ensure the correct actions were occurring in the Java code, and running a development version of the website while new features were being coded.

This was more important for the website, as an IDE was not being used. Therefore errors in the code might not be visible to the eye until the code has been run and tested manually. Eclipse

was able to provide more visible warnings and errors in the Java code before runtime. Using a Ruby IDE such as RubyMine may have helped with the testing, but ultimately the manual testing was done as a natural part of implementation, aided by tools where applicable.

3.3.2 Stress testing

Due to the nature of the system, a stress test was required to see how the system would handle a large amount of data being uploaded at once, and how fast results would be produced.

To test the system, 3 sets of student code was anonymised by Chris Loftus so that a realistic project could be tested in the system. For the stress test, these projects were then duplicated up to a total of 50 uploads. While this gives a good indication as to how fast the system might run, the actual checking speed may not be reliable, as code has been purposely duplicated to create a larger dataset. Therefore when checking, the system will detect that the projects are identical.

Uploading 50 files on the website did not stress the system at all, with all 50 projects being uploaded to S3 in under 3 seconds. Processing of these results began soon after this, and a few issues were found. As the free tier Elastic Compute Cloud (EC2) machines are only running 1 virtual CPU and 1GB of RAM, after processing a small number of submissions, Java memory exceptions were thrown. This was because for every check that is done between two files, a 2D integer array was being created, which could be quite sizeable depending on the size of the files. Therefore, with the environment that was available, the stress test was not able to be completed.

However, from what was seen within the stress test, the speed at which submissions were being checked was extremely fast, and if the memory exceptions were fixed in the future, speed should not be of any concern. To deal with these exceptions, a change of structure may be required, so that a more powerful machine could be run to deal with an assignment at a time. This structure could alter the system, so that an assignment would not be checked until a user says that all submissions have been uploaded to the assignment, and are ready to be checked. This removes the requirement of having a system looping over every available submission, and would reduce the number of duplicate checks being made in the software.

3.3.3 Comparing other software

As discussed in section 1.1.3, the Measure of Software Similarity (MOSS) application developed by Alex Aiken of Stanford University is the only program that also checked code for similarities. Although the project did not want to dwell on MOSS too much in case of copying ideas from it, towards the end of the project, the same test data that was used in the Stress Testing was sent to MOSS to compare the results.

To use MOSS, the first step was to sign up. Once this was done, a perl submission script was provided to upload submissions to their servers to be checked. This script required permissions to be changed, and to be placed in usr/bin on a Linux machine. The following command was then used to send the code:

```
moss -l Java -b Application.java -b ...java -d projectA/src/*.java
projectB/src/*.java
```

MOSS then returned the results seen in figure 3.1 below. These results differ quite significantly to the Code Collusion Checker's results. While the Code Collusion Checker calculated collusion results of around 24-27% between the three projects, MOSS only found a similarity of 1-9%. However, this was not surprising as the Code Collusion Checker was only checking against literal strings while taking into consideration the template file, whereas the MOSS project more likely took into consideration more variables than just the literal strings.

File 1	File 2	Lines Matched
xxx11_maps/map/src/ (6%)	xxx13_maps/src/ (9%)	68
xxx12_maps/src/ (3%)	xxx13_maps/src/ (5%)	54
xxx11_maps/map/src/ (1%)	xxx12_maps/src/ (1%)	15

Figure 3.1: Results produced by the MOSS system.

Chapter 4

Evaluation

4.1 Project requirements and feature comparison

The Code Collusion Checker was a project that aimed to created a system for lecturers that could check student code submitted to an assignment for collusion amongst the students. Methods that were specifically mentioned in the brief were string matching, and the possibility of using parsers.

Once the project began, the feature list was created to acquire a vision of what features might be required for the system, looking back at conversations with the supervisor and at the project brief. The feature list in Figure 1.1 is what was created for the project, with high level features to allow for flexibility of planning in the project, while still specifying everything that would be required. While having the high level features was advantageous from a developer's perspective so that not everything in the system was concrete, it may have been better to create a secondary feature list that went into more depth as to what might be required for each feature. This could have been created at the start of every iteration, and acted like short stories from an extreme programming (XP) approach. This would have helped the planning of the checking algorithms themselves a lot more, as there was a lot more work involved in the string matching than originally expected, which led to not enough time to implement a parsing checker. With more in depth planning of more complex features, it may have been possible to avoid some of the problems found throughout the project, to allow for more time for implementation of more features such as the parse checker.

Overall, the features were well thought out, and with the feature list being expanded throughout the project, it allowed for a continuation of development past the project deadline, for future developers to have something to expand upon. The requirements of the project were mostly met, with the exception of parse checking. However, it was not certain from the beginning of the project whether it would be manageable, as there was no prior experience with parsers coming into the project. While some research was done into what might be required, there was no implementation due to the time constraints of the project.

4.2 Methodology

Discussed in Section 1.2, an FDD-like approach was taken to the project, where adaptations had been made to suit a one person project. This choice was made due to the prospect of upfront design, and consistent iteration work. Furthermore, as the main project aims and an idea of the structure of the project was already known, the ability to create upfront design was possible. The methodology that was taken was followed well, with up front design at the beginning of iterations that built upon an overall model of the project.

An alternative approach that could have been taken would have been an XP-like approach. The approach that XP takes is to get as much value to the customer as quickly as possible. However, with not much design involved, it was not a methodology worth adapting, as the upfront design diagrams from FDD helped shape the project from the beginning. Furthermore, a lot of concepts from XP would not have worked for the project, such as an on-site customer or pair programming. Pair programming was physically not possible due to the project being a single-person project, and the idea of an on-site customer would not have been practical. While the supervisor did act as a customer for the software, it was not a case of being an on-site customer. An on-site customer is much more involved in a project, constantly suggesting new ideas to be added as stories to the project. While possible, it was more practical within the single-person project for the customer to be reviewing the product weekly and giving feedback in this manner, while not influencing the project to become something that it was not originally designed for.

The methodology was one of the stronger points of the project, as it helped keep the project on track through it's iterative development and design planning. In order to improve the methodology for future project, more of an emphasis on testing should have been made.

In standard FDD, testing should be part of every iteration to check that features that have been created function as desired. Despite this, the decision in the project was to not make potentially redundant tests while the structure of the code was constantly changing. This was not the correct approach to take, as it meant by the end of the project there were not many tests in the system to help future development of the project.

4.3 Design decisions

Following the FDD methodology taken, design was at the forefront of the project, to help shape the project from the beginning. Design diagrams were created during the first iteration to help direct spike work within the second iteration. More design diagrams could have been created to aid the project further, however it was believed that once the diagrams were created, they were sufficient for the project. While it is true that they aided development of the project, more diagrams would be needed to help new developers to the project understand the overall system that has been created.

From the beginning of the project, it was decided that the project would be developed for the cloud. Rather than develop the project to suit a local server before integrating it with cloud services, the project was designed to be created in the cloud from the beginning. In retrospective, this was not a good decision, as a lot of time was spent learning tools for Amazon Web Services (AWS), and learning how to get the application to communicate with the services created in the cloud. A more optimal approach would have been to create the areas of the application without taking into consideration the cloud aspect of the project, before migrating the code base to the

cloud to work alongside the services that are available. This approach was deemed unnecessary for the project however, due to the idea of coding a project in a direction that would ultimately be changed.

The overall design of the project was very successful, by keeping a simple website that was user friendly. By detaching the worker from the website, the website would never be under any pressure, and all load balancing in the future would be focusing on the worker itself. This design was ideal for the project, as due to the nature of assignments at university, the system would only be used after deadlines had been reached, where bulk amounts of code would need to be checked at the same time.

4.4 Tools and technologies used

Of all the tools and technologies used, the most important, but misused, tool was the version control on GitHub. The decision with version control was to keep the process simple, by only having a master branch and a development branch. All development would be completed on the development branch, and at the end of an iteration there would be a pull request to the master branch. However, a more ideal approach would have been to have branches for every iteration that was created, so that once the pull request was complete, the branch could be deleted. This complicated the overall version control on GitHub, but was manageable. Trying to push changes to GitHub regularly was the other issue in the project, leading to GitHub acting more as a back-up to the project rather than version control. While this was okay from a single person project perspective, a stricter approach should have been taken so that more regular updates would have been pushed to the repository. Instead, there were much larger commits being made on an irregular basis, due to wanting to complete features before pushing them to version control. To fix this issue, more management of the repository was required to assign features to their own branches. Ultimately, this thought process quickly overcomplicated what was required for the single person project.

As discussed in Section 4.3 the approach to developing the cloud based application was not ideal. Despite this, the decision to use the cloud was vital, as it helped address the issue that as more submissions would be checked, the time to check more submissions would exponentially increase, creating a time complexity to check an entire assignment's worth of submissions O(n!). The cloud helps solve this issue by providing the means of elastic scalability, to allow machines to scale upwards and outwards when under heavy workloads. The design of the system gave further time for checking to be completed, as a lecturer using the system would upload the submissions, and was able to leave the software to check the code, and report back at a later point the results.

Early in the project, there was a decision discussed in Section 1.1.2 about what languages should be used. The decision to use Ruby on Rails and Java over a C#.NET was the correct choice in hindsight, as other tools in the project complimented Java specifically well, and as a cloud project, using these languages on AWS was much less strenuous than it could be using C# outside of Microsoft's Windows Azure.

4.5 Final product

The final product that has been produced achieves a lot of the goals that the project aimed to achieve. By creating and implementing the entire workflow for the system, and checking code via string matching, the main goal that was not achieved was to implement parse checking. While this as disappointing, as it is the only part of the system that was not achieved, it was overall a successful product. Looking back at the project shows that attempting to implement parser checking in the time frame was out of scope in the time frame.

Using the Levenshtein Distance was very successful, and was not something that was thought to be required until research was completed in the area of string matching. One of the mistakes made in the project, which could have allowed for an attempt at implementing parser checking, was to try and create a new implementation of the Levenshtein distance. This was desired mostly because it was not ideal to be using third-party code. However, attempting to develop something more efficient was not a good decision, as the code originally used was very efficient and hard to improve upon. Given more time, the algorithm could be improved, perhaps through the use of a more efficient searching algorithm. For the scale of the project, this was not feasible.

Comparing the final product to the Measure of Software Similarity (MOSS) application, the project has been a great success. Although at the current time MOSS can be considered to be more accurate in it's calculations, the Code Collusion Checker has the potential to expand past MOSS, while keeping the usability that has been created in the project. This usability is what makes the project stand out against MOSS, and given more development, can easily replace MOSS as a collusion detector.

4.6 Future Work

When developed in the future, the following areas could be developed:

- Creating a parse checker to enable more accurate parsing would be vital. By following on from research completed in the final iteration (see Section 2.7.5), a parser could be made to create a flow of a program to try and determine similarities in how the method runs behind the literal text that the string checkers work against.
- Having results that could be exported outside of the system would help with the usability of the application, as it could be understandable that the results would be easier to understand within an Excel spreadsheet.
- While the software has been developed on the cloud, scalability would not be fully achievable in the current state of the software. The queue would have to be managed better, and a system put in place where submissions are not checked more than once, but to ensure that every submission is checked.
- The project aimed to achieve checking against Java code. This has been achieved, and as the software goes forward, it would need to be expanded to have the capability to check a wider range of languages.
- Being able to manage more versions of compressed files would also be desirable in the system, as not all projects are handed in as .zip files. Support for .tar.gz and other compression

formats should also be manageable.

- For a more long term ambition, integration with Blackboard would be the end goal, to eliminate the requirement for a front end application. Instead, Blackboard might be able to deliver the submissions directly to the Code Collusion Checker, and provide collusion results to the lecturer's via Blackboard.

4.7 Summary

When looking back over the project as a whole, there are a few areas that I would tackle differently.

During the initial stages of the project, spike work was completed to see what was achievable with the cloud. In the project, this took place over a week long iteration, where a workflow was created and tested in the cloud. If this spike work were to take place over a longer period, I believe that spike work into solutions for string and parser checking might have been possible. However, I would not want to spend longer than 2 weeks on this spike work for a project within this time frame, to ensure that iterative development could commence.

During the iterative development, testing was underrated and ignored for a large portion of the project, due to reasons discussed in Section 4.2. Emphasis was required on testing and refactoring in the project to try and help create a more maintainable code base for future developers.

Finally, version control through GitHub was not managed well. Starting the project again, a plan would be created regarding version control management, creating a version of the feature list on GitHub to reference, while regular updates via git would be required to ensure back-ups of the software are stored correctly.

Overall, I feel like the project was a success. The system was constructed in the cloud, and was able to complete basic collusion checking with template files in a fast manner. The project helped me learn more about how cloud computing works, and how a single person project can be managed using an agile methodology.

Appendix A

Third-Party Code and Libraries

1.1 Libraries

Ruby on Rails [6] - The website in the application was written using the Ruby on Rails framework, to create an MVC website. Ruby version 2.3.1p112 and Rails version 5.0.1 were used.

AWS SDK for Ruby [18] - To enable connectivity to the message queue hosted on the Simple Queue Service (SQS), and to store files on the Simple Storage Service (S3), the AWS SDK for Ruby was used. Version 2 of the SDK was used by adding the 'aws-sdk' gem to the gemfile.

Java Database Connectivity (JDBC) Driver [17] - To allow the Java worker to communicate with the database, a JDBC driver was required. The MySQL driver version 5.1.40 was used.

AWS SDK for Java [20] - For accessibility to SQS and S3, the AWS SDK for Java was required. Version 1.11.105 was installed through the AWS Toolkit for Eclipse.

Bootstrap [25] - To provide the website with helpful CSS and JavaScript functions, Bootstrap was used on the website. Version 3.3.7 was used.

Bcrypt [32] - When users were introduced into the system, some form of encryption was required for passwords. Ruby on Rails suggests to use the bcrypt gem for this, and version 3.1.7 was used for this.

1.2 Code

1.2.1 Levenshtein Distance

The first implementation of the Levenshtein Distance that was used in the software was based on the short essay Levenshtein Distance, in Three Flavors by Michael Gilleland [28]. The structure and functionality of the code was not changed. However, to ensure the code was readable and understandable, variable and method names were changed. Furthermore, the third party code only runs up to providing the actual Levenshtein distance, also known as the edit distance. From here, a percentage was calculated based on the length of the larger string, and the edit distance that was calculated.

```
private float getCollusionPercentage(String originalCode,
    String targetCode) {
    int cost;
    if(originalCode.equalsIgnoreCase(targetCode)){
        return 100;
    }

    comparisonMatrix =
        new int[originalCode.length() + 1][targetCode.length() + 1];

    for(int i = 0; i <= originalCode.length(); i++){
        comparisonMatrix[i][0] = i;
    }
    for(int i = 0; i <= targetCode.length(); i++){
        comparisonMatrix[0][i] = i;
    }

    for(int i = 1; i <= originalCode.length(); i++){
        char originalCharacter = originalCode.charAt(i-1);

        for(int j = 1; j <= targetCode.length(); j++){
            char targetCharacter = targetCode.charAt(j-1);

            if(originalCharacter == targetCharacter){
                cost = 0;
            }
            else{
                cost = 1;
            }

            comparisonMatrix[i][j] = findMinimum(
                    comparisonMatrix[i-1][j] + 1,
                    comparisonMatrix[i][j-1] + 1,
                    comparisonMatrix[i-1][j-1] + cost);
        }
    }

    int editDistance =
        comparisonMatrix[originalCode.length()][targetCode.length()];
    float collusionPercentage;

    if(editDistance == 0){
        collusionPercentage = 100;
    }
    else{
        int largerLength = originalCode.length();

        if(targetCode.length() > largerLength){
```

```
            largerLength = targetCode.length();
        }

        collusionPercentage = ((largerLength - editDistance) /
            (float)largerLength)*100;
    }

    return collusionPercentage;
}

private int findMinimum(int above, int left, int diagonal){
    int minimum = above;

    if(left < minimum){
        minimum = left;
    }
    if(diagonal < minimum){
        minimum = diagonal;
    }

    return minimum;
}
```

Appendix B

Ethics Submission

AU Status
Undergraduate or PG Taught

Your aber.ac.uk email address
sjn3@aber.ac.uk

Full Name
Stephen Norwood

Please enter the name of the person responsible for reviewing your assessment.
Reyer Zwiggelaar

Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment
rrz@aber.ac.uk

Supervisor or Institute Director of Research Department
cs

Module code (Only enter if you have been asked to do so)
CS39440

Proposed Study Title
MMP Code Collusion Checker - A system designed for lecturers to upload student code to a web application, that then checks the code for collusion between students in the assignment.

Proposed Start Date
30/01/2017

Proposed Completion Date
08/05/2017

Are you conducting a quantitative or qualitative research project?
Mixed Methods

Does your research require external ethical approval under the Health Research Authority?
No

Does your research involve animals?
No

Are you completing this form for your own research?
Yes

Does your research involve human participants?
No

Institute
IMPACS

Please provide a brief summary of your project (150 word max)
A system that is designed to allow lecturers to upload student code to a web application. The implementation stores this code on a private S3 server on AWS, before a Java application checks the code for collusion. This checking is done against other students who have also submitted work for the assignment. Anonymised code will be used as a test case in the system, which will be provided by Chris Loftus.

Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material?

Appendix C

Code Examples

3.1 Custom Levenshtein Distance implementation

As seen in Appendix A section 3.1, the original implementation of the Levenshtein Distance was taken from Michael Gilleland's implementation in his short essay Levenshtein Distance, in Three Flavors. Michael's method brute forces a matrix to find the edit distance between the two strings being compared. Once this was implemented, Feature #14 was created to try and improve on the algorithm, and to create a unique implementation of the Levenshtein Distance for the project.

This implementation attempts to cut down on the number of checks made against characters, to try and find the edit distance in a shorter time.

```
public float alternateLevenshteinChecker(String original, String target){
    if(original.equalsIgnoreCase(target)){
        return 0;
    }

    //Add null char to beginning to act as 0,0 in the matrix
    original = "0" + original;
    target = "0" + target;

    Integer[][] comparisonMatrix =
        new Integer[original.length()][target.length()];

    LinkedList<Tuple> queue = new LinkedList<Tuple>();
    ArrayList<Tuple> checkedPositions = new ArrayList<Tuple>();

    //Start at 0,0
    comparisonMatrix[0][0] = 0;
    queue.addFirst(new Tuple(0,0));

    Tuple current;
    int x, y, currentCost, cost;
    char originalChar, targetChar;
```

```
while(!queue.isEmpty()){
    current = queue.removeFirst();

    //Debug code
    //System.out.println("Checking (" + current.getX() + "," +
        current.getY() + ")");

    if(checkedPositions.contains(current)){
        //Debug code
        //System.out.println("Already checked");
        continue;
    }
    //Stops any looping of positions
    checkedPositions.add(current);

    x = current.getX();
    y = current.getY();

    if(current.getX() == original.length() - 1 &&
            current.getY() == target.length() - 1){
        //End of matrix found
        break;
    }

    try{
        currentCost = comparisonMatrix[x][y];
    }
    catch(ArrayIndexOutOfBoundsException ex){
        System.out.println("(" + x + "," + y +  ")
            Array Out of bounds");
        continue;
    }

    //Check diagonal
    try{
        originalChar = original.charAt(x + 1);
        targetChar = target.charAt(y + 1);

        if(originalChar == targetChar){
            cost  = 0;
        }
        else{
            cost = 1;
        }

        //Update matrix
        try{
```

```
            if(comparisonMatrix[x + 1][y + 1] == null){
                comparisonMatrix[x + 1][y + 1] = currentCost + cost;
            }
            else if(comparisonMatrix[x + 1][y + 1] >
                currentCost + cost){
                comparisonMatrix[x + 1][y + 1] = currentCost + cost;
            }
        }
        catch(ArrayIndexOutOfBoundsException ex){
            //Can continue here, as +1 +1 is out of bounds,
                so left and right will be affected
            System.out.println("Diagonal (" + (x + 1) + "," + (y + 1)
                 +  ") Array Out of bounds");
            continue;
        }
    }
    catch(StringIndexOutOfBoundsException ex){
        //Can continue here, as +1 +1 is out of bounds,
            so left and right will be affected
        System.out.println("Diagonal (" + (x + 1) + "," + (y + 1)
             +  ") String Out of bounds");
        continue;
    }

    if(cost == 0){
        queue.addFirst(new Tuple(x + 1, y + 1));
    }
    else{
        queue.addLast(new Tuple(x + 1, y + 1));
    }

    //Set cost to 1 for right and down
    cost = 1;

    //Update right of matrix
    try{
        if(comparisonMatrix[x + 1][y] == null){
            comparisonMatrix[x + 1][y] = currentCost + cost;
        }
        //If the current value of the matrix is more than the
            new cost, update
        else if(comparisonMatrix[x + 1][y] > currentCost + cost){
            comparisonMatrix[x + 1][y] = currentCost + cost;
        }

        queue.addLast(new Tuple(x + 1, y));
    }
    catch(ArrayIndexOutOfBoundsException ex){
```

```
            System.out.println("Right (" + (x + 1) + "," + (y)
                +  ") Array Out of bounds");
        }

        //Update bottom of matrix
        try{
            if(comparisonMatrix[x][y + 1] == null){
                comparisonMatrix[x][y + 1] = currentCost + cost;
            }
            //If the current value of the matrix is more than the
                new cost, update
            else if(comparisonMatrix[x][y + 1] > currentCost + cost){
                comparisonMatrix[x][y + 1] = currentCost + cost;
            }

            queue.addLast(new Tuple(x, y + 1));
        }
        catch(ArrayIndexOutOfBoundsException ex){
            System.out.println("Down (" + (x) + "," + (y + 1)
                +  ") Array Out of bounds");
        }
    }

    int editDistance =
        comparisonMatrix[original.length() - 1][target.length() - 1];
    float collusionPercentage;
    if(editDistance == 0){
        collusionPercentage = 100;
    }
    else{
        int largerLength = original.length();

        if(target.length() > largerLength){
            largerLength = target.length();
        }

        collusionPercentage = ((largerLength - editDistance)
            / (float)largerLength)*100;
    }

    return collusionPercentage;
}
```

3.2 Reading zip files from S3

The submissions of code were stored in zip files in the Simple Storage Service (S3) provided by Amazon Web Services (AWS). Downloading and extracting zip files from S3 was more challenging than expected, and when trying to apply template files to the submissions made this process even more complex.

```
public ArrayList<String> getFiles(String fileKey){
    ArrayList<String> fileStrings = new ArrayList<String>();

    S3Object object = s3Client.getObject(new GetObjectRequest(bucketName,
        fileKey));
    byte[] buffer = new byte[2048];

    ZipInputStream inputStream = new
        ZipInputStream(object.getObjectContent());

    ZipEntry entry;
    try {
        while((entry = inputStream.getNextEntry()) != null){
            if(!entry.isDirectory()){
                String fileName = entry.getName();
                if(!fileName.endsWith(".java")){
                    continue;
                }

                //Do not want subdirectories of files, only final name
                int lastSlashPosition = fileName.lastIndexOf("/");
                fileName = fileName.substring(lastSlashPosition + 1);

                File file = new File(uncompressedFileLocation
                    + "uncompressedFile");
                file.createNewFile();

                FileOutputStream fileOutput = new FileOutputStream(file);
                int len;
                while((len = inputStream.read(buffer)) > 0){
                    fileOutput.write(buffer, 0, len);
                }
                fileOutput.close();

                if(templateNames != null){
                    if(templateNames.contains(fileName)){
                        String diffCommand = "diff -u -b "
                                + uncompressedFileLocation
                                + fileName + " "
                                + uncompressedFileLocation
```

```
                        + "uncompressedFile | "
                        + "grep -E \"^\\+\" | "
                        + "sed -n '1!p' | "
                        + "sed -r 's .  ' > "
                        + uncompressedFileLocation
                        + "uncompressedFileDiff";
                System.out.println(diffCommand);

                String[] command = new String[]{
                        "/bin/sh",
                        "-c",
                        diffCommand
                };

                Process proc = Runtime.getRuntime().exec(command);
                try {
                    proc.waitFor(1, TimeUnit.MINUTES);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                file.delete();
                file = new File(uncompressedFileLocation
                    + "uncompressedFileDiff");
            }
        }
        String randomAccessFileName = uncompressedFileLocation
            + "uncompressedFile";
        if(templateNames != null){
            randomAccessFileName = uncompressedFileLocation
                + "uncompressedFileDiff";
        }
        RandomAccessFile randomFile = new
            RandomAccessFile(randomAccessFileName, "r");

        StringBuilder stringBuilder = new StringBuilder();

        String line;
        while((line = randomFile.readLine()) != null){
            stringBuilder.append(line);
        }

        randomFile.close();
        file.delete();
        //If file is empty (identical to template file),
            don't bother adding
        String fileString = stringBuilder.toString();
        if(!fileString.isEmpty()){
            fileStrings.add(stringBuilder.toString());
```

```
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
    return fileStrings;
}
```

Annotated Bibliography

[1] "The Levenshtein Algorithm," http://www.levenshtein.net, 2017.

A good image representation of the Levenshtein Distance, and a small block of text further describing how the algorithm works.

[2] "Heroku, Cloud Application Platform," https://www.heroku.com, 2017.

Heroku is a cloud service provider that was researched as an option to host the software on.

[3] "AWS Free Tier," https://aws.amazon.com/free/, 2017.

Amazon Web Services (AWS) offer a free tier of their cloud platform to help test features they offer.

[4] "Amazon Simple Storage Service (S3) - Cloud Storage," https://aws.amazon.com/s3/, 2017.

AWS Simple Storage Service (S3) provides object storage in the cloud with a web interface to communicate with.

[5] "Amazon Simple Queue Service (SQS) - Fully Manage Message Queues," https://aws.amazon.com/sqs/, 2017.

AWS Simple Queue Service (SQS) provides a message queue service in the cloud to allow communication between applications.

[6] "Ruby on Rails," http://rubyonrails.org/, 2017.

Ruby on Rails is a web framework used to create MVC websites in Ruby.

[7] "Sublime Text," https://www.sublimetext.com/, 2017.

Sublime is a multi-platform text editor for coding, and was used to develop the Ruby on Rails code for the project

[8] "Eclipse IDE for Java Developers," http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/neon3, 2017.

Eclipse is the IDE that was used to develop the Java code in the project

[9] "Amazon Relation Database Services (RDS)," https://aws.amazon.com/rds/, 2017.

AWS Relation Database Service (RDS) provides a service to help host databases in a safe environment

[10] "GitHub," https://github.com/, 2017.

GitHub provides a version control platform, where a repository was made to store the project code online

[11] Alex Aiken, "A System for Detecting Software Similarity," https://theory.stanford.edu/~aiken/moss/, 2017.

The Measure of Software Similarity (MOSS) software was the only piece of software found during research that was similar to the application being created in the project.

[12] "Ruby on Rails Security Guide," http://guides.rubyonrails.org/security.html#sql-injection, 2017.

Ruby on Rails provide a guide on security to explain countermeasures to common attacks including SQL Injection

[13] "Ruby on Rails Security Guide," http://guides.rubyonrails.org/security.html#cross-site-scripting-xss, 2017.

Ruby on Rails provide a guide on security to explain countermeasures to common attacks including Cross Site Scripting (XSS)

[14] J. Hunt, *Agile Software Construction*. London : Springer, 2006, pp. 31–45.

Chapter 3 of Agile Software Construction talks about how Agile Modelling can be acheived to aid with agile development. Chapters 9 and 10 (pages 161-183) were also read to aid with FDD.

[15] "AWS Elastic Beanstalk - Deploy Web Applications," https://aws.amazon.com/elasticbeanstalk/, 2017.

AWS Elastic Beanstalk (EB) helps automatically deploy code to an EC2 instance to host web applications

[16] "AWS Elastic Beanstalk Developer Guide," https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/, 2017.

Guides on the Elastic Beanstalk Developer Guide website helped set up and host the website on AWS.

[17] "MySQL Download Connector," https://dev.mysql.com/downloads/connector/j/5.1.html, 2017.

The official download page for the JDBC driver used to connect to the MySQL database in Java

[18] "AWS SDK for Ruby V2," http://docs.aws.amazon.com/sdkforruby/api/, 2017.

The AWS SDK for Ruby used in the Ruby on Rails website has an online API to help with development in the early stages, with connecting to SQS and S3

[19] Maurcio Linhares, "Make the most of SQS in Ruby," https://mauricio.github.io/2014/09/01/make-the-most-of-sqs.html, 2014.

A guide that explains how to communicate with SQS in Ruby

[20] "AWS SDK for Java," http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html, 2017.

The AWS SDK for Java used in the worker has an online API to help with development in the early stages, with connecting to SQS and S3

[21] "Receiving and Deleting a Message from Amazon SQS Queue," http://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-receive-delete-message.html, 2017.

A guide by Amazon Web Services that explains communication between SQS and Java

[22] "Upload an Object Using the AWS SDK for Ruby," http://docs.aws.amazon.com/AmazonS3/latest/dev/UploadObjSingleOpRuby.html, 2017.

A guide by Amazon Web Services that explains how to upload files to S3 through Ruby

[23] "Get an Object Using the AWS SDK for Java," http://docs.aws.amazon.com/AmazonS3/latest/dev/RetrievingObjectUsingJava.html, 2017.

A guide by Amazon Web Services that explains how to retrieve a file from S3 to Java

[24] "Elastic Compute Cloud (EC2) - Cloud Server and Hosting," https://aws.amazon.com/ec2/, 2017.

AWS Elastic Compute Cloud provides virtual machines, which was used to host the Java worker

[25] "Bootstrap," http://getbootstrap.com/, 2017.

Bootstrap is a HTML, CSS and JavaScript framework that provides a useful library of code to help develop a quick website

[26] "Design Patterns in Java Tutorial," https://www.tutorialspoint.com/design_pattern/index.htm, 2017.

While researching different design patterns that may be required for the worker, this tutorial helped explain and gave examples of how to code different patterns in Java.

[27] V. Levenshtein.

The original achedemic paper explaining the theory behind what is now known as the Levenshtein Distance

[28] M. Gilleland, "Levenshtein Distance, in Three Flavors," http://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm, 2017.

A short essay describing how the Levenshtein Distance works, complete with a working example that was used as a basis for the original implementation of the Levenshtein Distance.

[29] "AWS Lambda - Serverless Compute," https://aws.amazon.com/lambda/, 2017.

AWS Lambda provides a service that runs code on demand, through the use of triggers

[30] "AWS Push Notification Service - Amazon SNS," https://aws.amazon.com/sns/, 2017.

AWS Simple Notification Service (SNS) provides a push notification service that other services can listen to

[31] "AWS Toolkit for Eclipse," https://aws.amazon.com/eclipse/, 2017.

The toolkit for Eclipse allows for easy development and deployment of AWS projects, such as a Lambda project

[32] "Bcrypt," https://github.com/codahale/bcrypt-ruby, 2017.

Bcrypt was the encryption library used on Ruby on Rails to create secure passwords for the wbesite

[33] G. Buckner, "Simple Authentication with Bcrypt," https://gist.github.com/thebucknerlife/10090014, 2014.

A tutorial that guides through the basic use of Bcrypt in Ruby on Rails

[34] A. Alexander, "Running system commands in Java applications," http://alvinalexander.com/java/edu/pj/pj010016, 2016.

A guide talking about how to use Linux commands in Java, which was required to use the diff command to apply templates to submissions

[35] "JavaCC - The Java Parser Generator," https://javacc.org/, 2017.

JavaCC is a parser generator for Java that was researched to be used as part of the parse checkers

[36] "ANTLR," http://www.antlr.org/, 2017.

ANother Tool for Language Recognition (ANTLR) is a parser generator that was researched to be used as part of the parse checkers

[37] "JUnit 4," http://junit.org/junit4/, 2017.

JUnit4 was used as the Java unit testing framework

[38] "Cucumber-rails," https://github.com/cucumber/cucumber-rails, 2017.

Cucumber testing was used for automated acceptance testing of the website

[39] "Ruby on Rails Guides," http://guides.rubyonrails.org/, 2017.

A number of guides provided by Ruby on Rails were followed to aid development of the website