

# Investigating Software Quality on GitHub Projects

A dissertation submitted in partial fulfilment of  
the requirements for the degree of  
BACHELOR OF ENGINEERING in Computer Science  
in  
The Queen's University of Belfast  
by  
Gary McPolin  
29<sup>th</sup> April 2019

## **Acknowledgements**

I would like to acknowledge the supervisor of this project, Dr. Desmond Greer for his valuable support and guidance throughout completion of this dissertation. Additionally, to all lecturers in Queen's University, Belfast who provided me with the knowledge and practical skills required for completion of this project and application. Finally, thanks to Maurício Aniche, who developed the CK Metrics calculation tool used during this project.

## **Abstract**

GitHub has become the most popular web-based version control platform for open-source software, and the publicly-available GitHub API can be used to obtain information about a project and its contributors. Data can be extracted from the API which provides insight into the software development process and can be used to investigate how various factors affect code quality; for example, code change-frequency. This project employs object-orientated quality metrics to determine an overall measure of code quality for open-source Java projects. Coupled with information extracted from the GitHub API, this project further investigates the influential factors of code quality. A purpose-built tool has been developed which facilitates the investigation, allowing users to download GitHub projects, analyse their code quality and visualise the factors that influence quality. Four hypotheses were proposed regarding the effect of time, code size, change-frequency and number of contributors on code quality. Three of the four hypotheses were discovered to be valid. The results show that increasing time and code size negatively affect code quality, with increasing code size having the strongest relationship with loss of quality. Previous studies which found that the number of contributors has no significant effect on quality were corroborated. The investigation into the influence of change-frequency was inconclusive and provides an opportunity for further study.

## Contents

1.0	Introduction and Problem Area .....	1
2.0	Solution Description and System Requirements .....	3
2.1	Calculating Source Code Quality .....	4
	CK Metrics .....	4
2.2	Investigating Influential Factors on Code Quality .....	8
	Time .....	8
	Code Size.....	9
	Number of Contributors .....	9
	Change-Frequency .....	9
2.3	Project Selection .....	9
2.4	Application .....	10
2.5	Overview of Features (Use Cases) .....	12
2.6	Non-Functional Requirements.....	15
2.7	Interfaces .....	17
2.8	User Characteristics .....	18
3.0	Design.....	20
3.1	Application Architecture .....	20
3.2	User Interface Design .....	21
3.3	Software System Design.....	25
3.4	Database.....	26
3.5	Concurrency.....	28
3.6	Fault Tolerance and Error Handling .....	28
4.0	Implementation.....	29
4.1	Implementation Decisions .....	29
4.2	Software Libraries .....	29

4.3	Component Implementation .....	30
	Downloading Source Code.....	30
	Analysing Code Quality .....	31
	Code Climate.....	32
	Extracting GitHub Statistics.....	32
5.0	Testing.....	33
5.1	Unit Testing .....	33
5.2	Acceptance Testing.....	34
6.0	System Evaluation and Experimental Results.....	36
6.1	Code Quality over Time .....	36
6.2	Code Quality vs Code Size .....	37
6.3	Code Quality vs Change-Frequency .....	38
6.4	Code Quality vs Number of Contributors.....	39
6.5	Conclusion.....	40
7.0	References .....	42
8.0	Appendix .....	44

## 1.0 Introduction and Problem Area

Software quality is defined as the extent to which a system conforms to its specified requirements and fulfils the needs of its users. Many factors contribute to software quality; perhaps most significantly the quality of the source code. Source code quality is an approximation of its complexity, reusability, maintainability and testability. High-quality code should be coherent, thoroughly-tested and have the potential for future modification or re-use. Spinellis states that “quality, time and cost are the three central factors determining the success or failure of any software project” [1]. Naturally, low-quality software requires more substantial financial and temporal investment, whether this is in the scope of fixing bugs or repaying technical debt. Therefore, we can infer that two of these three factors are influenced by quality, justifying its importance as a determinant of successful software.

Extensive research has been directed towards determining which strategies are most effective in improving software quality, with the majority of research centred around two concepts; streamlining the development process and improving the quality of the source code. In recent years, IT companies have been adopting methodologies that aim to improve both of these aspects. Agile methodologies, for example, focus on the development process, while Extreme Programming (XP) and Test-Driven Development (TDD) aim to improve source code quality. Nevertheless, the intended outcomes of each methodology are consistent; to improve the effectiveness, maintainability and portability of the source code. With regards to enhancing the development process, findings regarding the most effective approach are contested and largely dependent on the nature of the software [2]. The primary focus of this project will be to investigate how aspects of the development process impact on code quality, where code quality is a product of individual quality aspects, such as complexity and reusability.

When considering the attributes of high-quality code, adherence to specified design and implementation principles is particularly important. However, there are numerous programming paradigms, each with differing guidelines. Object-Orientated Programming has become the most commonly used model since its conception 50 years ago [3]. Central to OOP is the use of its core features: inheritance, abstraction, encapsulation and polymorphism. The design and implementation principles of OOP are well-documented, and investigating the benefits of writing source code which adheres to them has been the subject of various research efforts.

Chidamber & Kemerer first proposed a suite of metrics for calculating individual aspects of the object-orientated approach in 1991 [4]. They developed six measures of compliance with the object-orientated design principles with the purpose of providing a method of evaluating the

quality of the software development process. The metrics attempt to uncover design violations; including complexity and lack of reusability; on a per-class basis. Although the CK metrics do not define thresholds for object-orientated design violations, the metric values can be assessed for outlying values. Outliers have been used to indicate rework, design and maintenance effort, as well as fault-proneness and productivity [5], [6].

The open-source software development platform, GitHub, has become the most commonly used web-based version control tool in the IT industry. At the time of writing, GitHub claims to host over 100 million repositories [7]. The site is primarily used to host software during and after its development. As a distributed version control tool, GitHub retains previous versions of software which provides the opportunity to analyse code quality over time or developmental iterations, making the platform an excellent source of data for this investigation. GitHub has developed a publicly-available API which facilitates the extraction of information which provides some insight into the development process. This data will be used to study the impact of developmental factors such as change-frequency and number of contributors on code quality.

This project will consider code quality to be a product of four attributes which are determinants of high-quality software. The attributes under consideration are complexity, reusability, testability and maintainability. The nature of software development and the pressure to deliver a product with a short time-to-market means that trade-offs in quality attributes are often made in favour of a shorter development cycle. Developing an understanding of how aspects of the development process affect code quality is significant for all parties involved in the software development cycle, both in the design and development phases. If it is known that particular development processes are more likely to lead to higher-quality code, which then translates into successful software, better implementation decisions can be made during the development process.

## 2.0 Solution Description and System Requirements

A single research question has been formed to provide direction for this project and define the investigations that the project aims to explore. Multiple hypotheses have been proposed that examine individual aspects of the research question in more detail. Each hypothesis for research will attempt to establish a relationship between code quality and a single factor from time, code size, change-frequency and number of contributors. A justification accompanies each hypothesis below.

**RQ1.** What is the impact of time, code size, change-frequency and number of developers on code quality?

**H1.** Code quality generally decreases over time.

**Justification:** The quickly-evolving nature of technology means that the lifespan of software is often relatively short. The technology used to develop software is updated and improved frequently, resulting in the development of new software which provides similar functionality but further implements the new and improved technology. The result is that software becomes outdated quickly. This hypothesis assumes that outdated software will attract less attention and maintenance, and thus, code quality will decrease.

**H2.** Code quality decreases as code size (LOC) increases.

**Justification:** A larger code size means more lines of code that have to be maintained, thus making maintenance more difficult. It is expected that this additional effort will result in less effective maintenance which will negatively impact code quality. Furthermore, the number of lines of code has been found to have a near-perfect linear relationship with McCabe's Cyclomatic Complexity [8]; as code size increases, so too does complexity. Therefore, as complexity is an indicator of low-quality, it is expected that code quality will decrease as code size increases.

**H3.** Code quality is greater in projects that are changed frequently.

**Justification:** Frequent commits to a repository indicate that the code is still being maintained which is expected to maintain or improve the code's existing quality. In contrast, code that is not changed frequently is likely to be obsolete and changes made are likely to accumulate technical debt and design issues, resulting in reduced quality.



**H4.** There is no relationship between code quality and the number of contributors.

**Justification:** A similar study, conducted by Norick et al., investigated the effect that the number of contributing developers had on code quality for open-source C++ projects [9]. They concluded that there was no correlation between the number of developers and code quality. The study used alternative code quality metrics than this project. However, the outcome is expected to be consistent.

The methodology for the project can be sub-divided into two parts: calculating source code quality and investigating factors that influence code quality. GitHub is the largest repository for open-source software and will be used to obtain the dataset for the project.

## **2.1 Calculating Source Code Quality**

There are numerous metrics available for calculating source code quality in relation to compliance with design and implementation patterns. This project will primarily use Chidamber & Kemerer's CK Metrics suite to evaluate Java source code with regards to the object-orientated design principles. A 2018 study found Java to be the most widely used language for enterprise application development [10]. Furthermore, according to [11], Java is the most popular object-orientated programming language on GitHub, thus justifying its selection as the focal point of this investigation.

### **CK Metrics**

The CK Metrics suite consists of 6 individual metrics calculated for each class. Chidamber & Kemerer's metrics aim to measure class-level conformity with aspects of the object-orientated approach. Numerous studies have concluded that various combinations of the CK Metrics are valid indicators of complexity, maintainability, reusability and testability [6], [12]. Five of the six metrics will be used in this project. NOC has been excluded due to the difficulty involved in calculating a value. **TABLE 1** provides a summary of each metric.

**TABLE 1: Chidamber & Kemerer CK Metrics Suite** [4], [13]

Abbr.	Metric	Description
CBO	Coupling Between Objects	A count of the number of couples with other classes. CBO can be understood to be the number of classes that a given class uses plus the number of classes used by the given class.
WMC	Weighted Methods Per Class	The sum of the complexities of methods in a class. $WMC = n$ for $n$ methods in a class if all method complexities are considered to be unity.
DIT	Depth of Inheritance Tree	The maximum path from the given class to the root class of the inheritance tree.
NOC	Number of Children	The number of immediate descendants of a class.
RFC	Response For a Class	The cardinality of the set of methods that can potentially be executed by an object of the class.
LCOM	Lack of Cohesion in Methods	A measure of the similarity between methods of a class and its instance variables.

This project will investigate source code quality with regards to four aspects of software development: complexity, reusability, maintainability, testability. These aspects can be measured using combinations of metrics from the CK Metrics suite. **TABLE 2** provides an overview of the mapping between CK Metrics and the factors associated with code quality.

**TABLE 2: Mapping Between CK Metrics and Code Quality Aspects** [12]

		Impact			
Metric	Value	Reduced Complexity	Increased Reusability	Increased Maintainability	Increased Testability
CBO	Low	✓	✓	✓	✓
WMC	Low		✓	✓	
DIT	High		✓		
	Low	✓			
RFC	Low	✓			✓
LCOM	Low	✓			

As mentioned previously, thresholds are undefined for CK metrics. However, aspects of the object-orientated approach can be measured by observing the metric values in comparison with the objectives for each quality aspect. Depending on the individual metric, a high or low value is optimal. For example, low CBO is an indicator of low complexity and high reusability [14]. However, the DIT metric is different in nature. Classes with high DIT are more likely to inherit a larger number of methods which increases the potential for re-use at the expense of

added complexity. In contrast, a low DIT score implies a simpler design with the effect of a reduction in complexity coupled with less potential for re-use. Therefore, the DIT metric involves a trade-off between complexity and reusability.

Alternatively, threshold values for each of the CK Metrics, defined by two separate studies, can be used. Shatnawi used logistic regression to identify threshold values for CBO, WMC and RFC and found that the selected threshold values were more accurate than those defined using data distribution parameters [15]. The remaining metrics, DIT and LCOM, were evaluated in a 2012 study into identifying threshold values for object-orientated software metrics. Ferreira et al. defined metrics for six object-orientated quality metrics, two of which were DIT and LCOM from the CK suite. Their study involved the analysis of 40 open-source Java programs. They concluded that a threshold value for LCOM could be derived from the heavy-tailed model of distribution. A threshold value for DIT was achieved by using the Poisson distribution to model DIT values [16]. The threshold values obtained from both studies are displayed in **TABLE 3**.

**TABLE 3: Threshold Values for CK Metrics [15], [16]**

<b>Metric</b>	<b>Threshold</b>
CBO	9
WMC	20
DIT	2
RFC	40
LCOM	0

This project will employ both methodologies mentioned above to determine scores for each of the quality aspects. Data from open-source Java projects will be analysed on a per-class basis in order to generate a dataset of scores for each metric in the CK suite for each class in the project. The dataset will then be assessed against the threshold values in **TABLE 3**.

The extent of deviation of the classes above the threshold values will be aggregated and scaled by the total number of classes to define a value for conformity with each metric. The deviation values are used to allow differentiation between projects that marginally violate the thresholds from projects which do so significantly. Project size is accounted for by considering the conformity values in relation to the number of classes in the project. In other words, we will calculate the extent to which classes violate the threshold for each metric, in comparison with the total number of classes in the project. The score is scaled based on the number of classes in

the project, such that the maximum conformity score is 0 and the minimum conformity score is  $-n$  where  $n$  is the number of classes in the project. Thus, each metric can achieve a maximum score of 0 if no classes exceed the threshold value. This formula will often score negatively due to the fact that it measures deviance from desired values, representing the effect of non-conformity on code quality. **FIGURE 1** displays the formula used to calculate each of the CK metrics on the project-level.

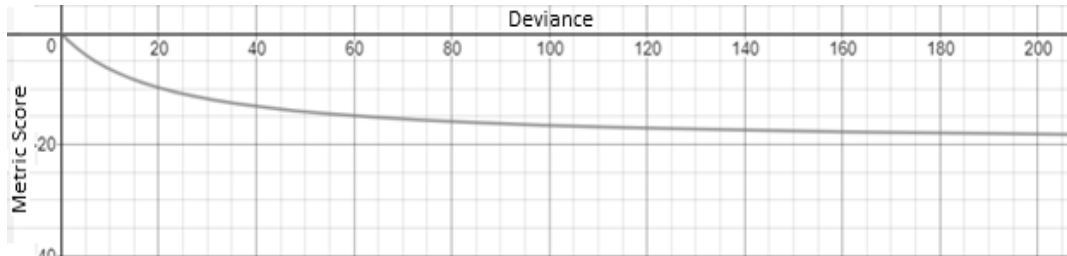
**FIGURE 1: Calculating Project-level CK Metrics**

*The formula below applies to each metric in the CK suite. Let  $d_1, \dots, d_n$  be the set of deviances of each class above the threshold value, i.e. the distance of each class's metric value from the threshold value for each class that exceeds the threshold. Let  $n$  be the total number of classes in the project. A constant value (+1) is added to the denominator to prevent possible division by 0. This constant does not significantly affect the result regardless of inputs.*

$$\text{Metric} = - \left( \frac{n * (\sum d_1, \dots, d_n)}{(\sum d_1, \dots, d_n) + n + 1} \right)$$

**FIGURE 2: Graph modelling individual CK Metric scores**

*The graph below shows the range of possible metric scores for each of the CK metrics where the number of classes = 20. The maximum value for each metric is 0 where there is no deviance and the minimum value is close to the asymptote at -20 where there is high deviance.*



Once a value for each metric has been calculated, an overall quality score can be derived using a weighted formula. The weightings used in this formula are based on the mappings in **TABLE 2** and are determined by the number of quality aspects that are affected by individual quality metrics. Optimal metric values have a total of 9 effects on quality aspects; denoted by “ticks” in **TABLE 2** and reduced by two due to the offsetting effects of high and low DIT scores. For example, the *coupling between objects* metric is a contributing factor in four quality aspects: complexity, reusability, maintainability and testability. Therefore, CBO is assigned a weighting of 4/9 (~0.44) in the quality formula. The *Weighted methods per class* metric affects two aspects and is therefore assigned 2/9 (~0.22) weighting. The quality formula is defined in **FIGURE 3**.

**FIGURE 3: Calculating Quality**

Let  $n$  be the total number of methods in the project. Let  $CBO$ ,  $WMC$ ,  $DIT$ ,  $RFC$ ,  $LCOM$  be the values derived from the equation in **FIGURE 1**. Note that the sum of the numerators for metric weightings adds to 11. However, the offsetting +/- weightings for  $DIT$  scores result in an actual sum of 9, hence the denominator.

$$\frac{n + \frac{4}{11}(CBO) + \frac{2}{11}(WMC) + \frac{1}{11}(DIT) - \frac{1}{11}(DIT) + \frac{2}{11}(RFC) + \frac{1}{11}(LCOM)}{n}$$
$$\therefore \text{Quality} = \frac{n + \frac{4}{9}(CBO) + \frac{2}{9}(WMC) + \frac{2}{9}(RFC) + \frac{1}{9}(LCOM)}{n}$$

One notable characteristic of the quality formula is that the  $DIT$  metric appears twice in the calculation, once with a positive weighting and once with a negative weighting of the same magnitude. This occurs because  $DIT$  has the potential to both positively and negatively affect quality. A high  $DIT$  score improves reusability, while a low  $DIT$  score reduces complexity.  $DIT$  has one positive and one negative effect on two quality aspects as outlined in **TABLE 2**, and therefore the  $DIT$  score positively and negatively affects quality with the same magnitude, resulting in the  $DIT$  score becoming redundant in this equation.

The formula in **FIGURE 3** could be modified in to favour reusability over complexity by redistributing the  $DIT$  weightings such that the positive  $DIT$  value improves the quality score more significantly than the negative  $DIT$  value decreases the score. This would have the effect of increasing quality with higher  $DIT$  scores, thus favouring reusability over complexity. Note that other weightings would also have to change to reflect this accurately.

## 2.2 Investigating Influential Factors on Code Quality

Now that code quality can be calculated; we can begin the investigation that is the primary objective of this project. This investigation aims to evaluate how code quality changes under different factors. This project will examine the effect of four separate factors on code quality: time, code size, number of contributors and change-frequency. Analysis of these metrics is made possible by GitHub.

### Time

Source code quality over time is the starting point of this investigation. This is a basic investigation that explores how code quality fluctuates over time, regardless of any other factors. Previous versions of a product can be downloaded from a GitHub repository, and source

code quality values can be determined. The values will be plotted on a graph to visualise the change in code quality over time.

### **Code Size**

Code Climate can be used to obtain data about repositories that is not available from GitHub and code size is one of the attributes available. This project will involve gathering versions of GitHub projects, analysing the code quality and ordering them by code size in lines of code. This will provide a visualisation of how code quality is affected by code size.

### **Number of Contributors**

One of the reasons that platforms such as GitHub are so popular is that they allow owners of software to open their product up to outside development. GitHub allows users to contribute to open-source projects; subject to the owner's approval; and also keeps track of the number of users who have contributed to each product. This data is available via the GitHub API. This project will examine the code quality of a number of projects from GitHub, with varying numbers of contributors, in order to determine whether there is a correlation between code quality and the number of developers on a project.

### **Change-Frequency**

Finally, this project will assess how the frequency of code changes impact quality. This can be measured by examining the frequency of commits for a number of GitHub projects against the quality of those projects. A scatter graph can be plotted which visualises average code quality against average change frequency for a number of projects, in order to determine whether there is a correlation between the two attributes.

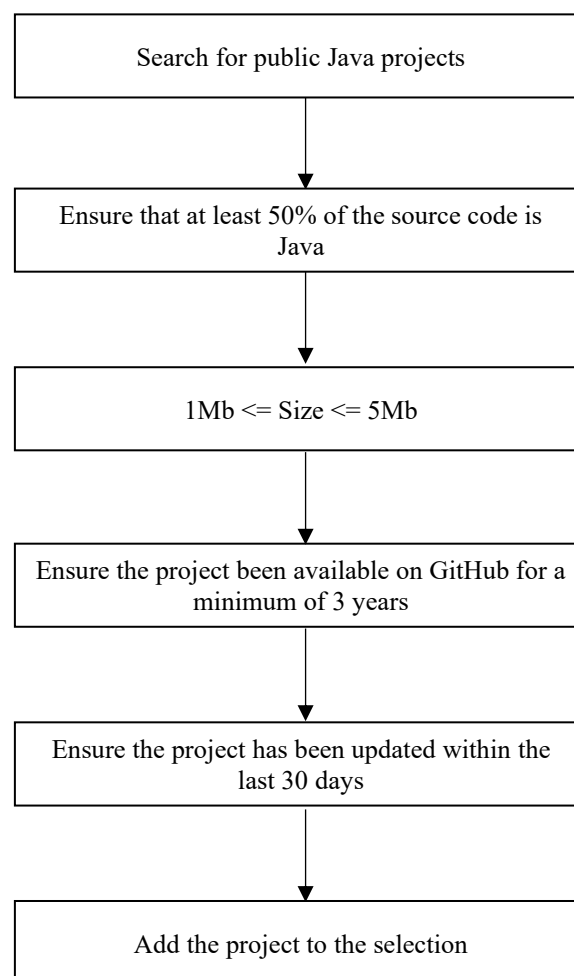
## **2.3 Project Selection**

The project selection process is outlined in **FIGURE 4**. Java projects are used exclusively in this project due to the popularity of the language and the limitations of the tools used to calculate the object-orientated CK Metrics. Therefore, the first step in project selection is to filter by the language used. In order to ensure the validity of our quality measure, projects selected should have a 50% affinity to Java. This is required as software projects often involve a mixture of programming languages. In order for our quality calculation to be accurate, we must analyse the quality of the primary development language.

Projects are limited to between 1Mb and 5 Mb in size in order to ensure that analysis could be completed in a reasonable amount of time, while ensuring that the projects are large enough to have sufficient code to analyse.

As most of the investigations involve metrics that could be influenced by time, the length of time projects have been available on GitHub must also be considered. For example, newly-created projects will be likely to have insufficient data regarding aspects of the development process, such as change-frequency. Therefore, this project will only consider projects that have had sufficient time to experience significant development in an attempt to negate possible imbalances. The threshold for project lifespan is a minimum of 3 years. Only projects which have been updated within 30 days of the investigation were considered in order to avoid projects which contain obsolete and unchanged code.

**FIGURE 4: Project Selection Process**



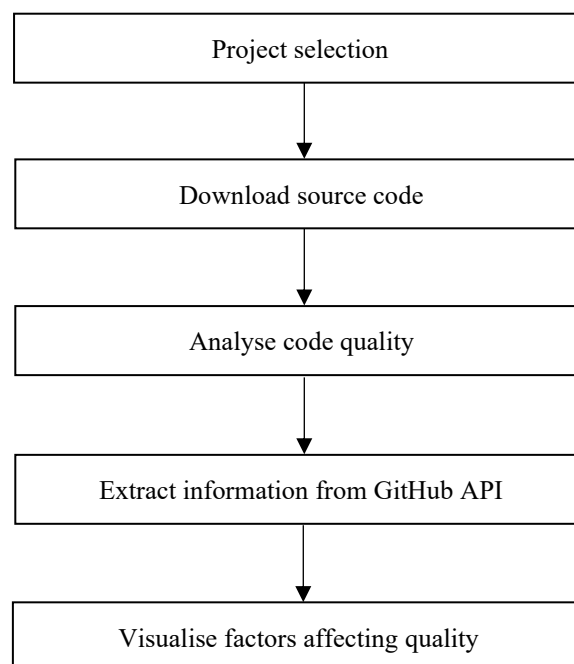
## 2.4 Application

A purpose-built web-application will be developed which will be designed to facilitate each aspect of the investigation demonstrated in **FIGURE 5**. However, the application should also be built with the intention of being reusable; and therefore, should offer the same

functionality for user-provided repositories. For example, users should be able to supply a URL link to one of their repositories, choose a suitable time period over which to conduct the investigation and view the results of the investigation in a user-friendly format. This functionality presents users with the opportunity to use the application to understand more about the quality of their code and the potential steps that could be taken to improve quality, based on trends from high-quality projects.

As the application will be primarily built to analyse code quality in object-orientated Java projects, the majority of code quality metrics cannot be calculated for other languages. This applies to all five of the CK Metrics and therefore complexity, reusability, maintainability and testability. However, technical debt can be calculated regardless of the primary programming language using an external tool [17]. The information extracted from the GitHub API is also independent of programming language and can be obtained for non-Java projects. Technical debt suffices as a substitute for code quality, meaning that the application remains useful for analysing quality regardless of the language used.

**FIGURE 5: Investigation Process**





## **2.5 Overview of Features (Use Cases)**

### Use Case 1

*Use Case Name:* User Login

*Description:* The user attempts to log in to the application using their GitHub credentials. The user is authenticated via OAuth, meaning that they are not required to register an account directly with the application. The user is prompted to grant permission to the application to access their GitHub data. Permission is required for access to the GitHub API and the user's personal repositories.

### Use Case 2

*Use Case Name:* Visualise GitHub Statistics

*Description:* The user provides one or more GitHub repository URLs to the */statistics* route and is presented with a summary of the data that can be extracted from the GitHub API. This functionality allows users to compare their repository to others in terms of the number of open issues, stargazers, branches, among other attributes.

### Use Case 3

*Use Case Name:* View trending repositories on GitHub

*Description:* The */trending* route displays a list of the repositories that are classified as trending by GitHub based on stars, forks, commits, follows and page views. This functionality allows users to view a list of repositories which are attracting lots of interest. Users can filter the list by a specific programming language and time period: today, this week or this month. A shortcut will also be provided to copy the URL for any trending repository so that it can be provided for analysis.

### Use Case 4

*Use Case Name:* Access Quick-Links to User Repositories

*Description:* The */repositories* route displays a list of the GitHub repositories belonging to the current user. From here, the user can copy the repository URL to the clipboard so that it can be submitted for analysis.

### Use Case 5

*Use Case Name:* Find GitHub repositories for analysis

*Description:* Users should be able to open a page which displays a list of GitHub repositories matching pre-defined search criteria. Default criteria should be provided which the user should be able to modify. Users should be able to configure the target language, minimum and maximum size, project creation date and last-updated date. The default parameters should display Java repositories ranging from 1Mb-5Mb that were created at least three years ago and have been updated within the last month. The purpose of this functionality is to provide a method of retrieving a list of projects based on specified criteria for analysis. Users should be able to quickly copy the URLs of some or all of the repositories so that they can be analysed.

#### Use Case 6

*Use Case Name:* Find GitHub repositories for analysis using custom search criteria

*Description:* The user should be able to change the default search criteria used on the discover page to find GitHub projects which meet their specific requirements. The user should be able to click on a link on the discover page which allows them to change any of the criteria used for the search, namely the target language, maximum and minimum repository size, minimum date of creation and the minimum date of last update. Clicking the submit button should result in the search being conducted using the new criteria, and the discover page will display the list of repositories that match the criteria.

#### Use Case 7

*Use Case Name:* Analyse Source Code for a single version of one repository

*Description:* The user provides a GitHub repository URL to the form located on the application landing page and uses the date picker to select a single date for which to download and analyse the source code quality for the repository as it was on the chosen date. The intention is that this functionality is primarily used to analyse code quality for a repository in its current state.

#### Use Case 8

*Use Case Name:* Analyse Source Code for a single version of multiple repositories

*Description:* This use case is largely identical to Use Case 7, except that the user can provide multiple GitHub repository URLs to be analysed simultaneously. The purpose of this functionality is to provide users with the ability to perform analysis on different repositories at the same point in time so that they can be compared to one another.

#### Use Case 9

*Use Case Name:* Analyse Source Code for multiple versions of a single repository

*Description:* This use case is largely identical to Use Case 8. The difference is that, rather than comparing multiple repositories, the user can analyse a single repository over a chosen time period. In order to do this, the user uses the date picker to choose the period over which to analyse a total of five versions of the project. Each version of the project is evenly-distributed across the time period. This functionality allows the user to compare changes in code quality for a specific project over time, code size, change-frequency and number of contributors.

#### Use Case 10

*Use Case Name:* Analyse Source Code for multiple versions of multiple repositories

*Description:* This use case combines Use Case 8 and Use Case 9. The user provides multiple GitHub repository URLs and uses the date picker to choose the period over which to analyse five evenly-distributed versions of each repository. This functionality allows the user to compare changes in code quality across multiple projects under the influence of time, code size, change-frequency and number of contributors in order to identify trends in factors which influence code quality.

#### Use Case 11

*Use Case Name:* Visualise code quality and GitHub statistics for multiple versions of a single project

*Description:* Having already analysed the quality of multiple versions of a project, the user can visualise changes in quality over each iteration of a project. A graph comparing code quality over each version is displayed. The user will also be provided with visualisations of data extracted from GitHub and Code Climate such as code quality vs the number of issues and technical debt.

#### Use Case 12

*Use Case Name:* Visualise code quality and the factors affecting it for multiple versions of multiple projects

*Description:* This use case allows the user to examine the factors affecting code quality across multiple projects simultaneously in order to identify trends. For example, looking at code quality for multiple projects in comparison with changes in the number of lines of code may identify that code quality generally decreases with the addition of more code. This use case is

the basis of the research conducted for this project. Statistical analysis is performed on each dataset to provide validation to apparent correlations.

#### Use Case 13

*Use Case Name:* Change the default language of a repository (linguist-language)

*Description:* GitHub will automatically determine the primary language of a project based on the total lines of code for each of the different programming languages used in the project. However, this assertion can be incorrect in some cases, for example, where a sizeable external library is included as part of a project. Some of the algorithms used to analyse source code in this project assume this value to be correct and therefore may perform analysis on a secondary language, thus leading to an inaccurate quality score. This functionality alerts the user to the assumed primary language of the repository and provides the necessary functionality to change it if it is incorrect.

#### Use Case 14

*Use Case Name:* View specific issues that are reducing code quality

*Description:* In addition to assigning a score for code quality to a project, the application should alert users to some of the issues in their code that are responsible for a non-perfect quality score. These issues are known as code smells. The user should be able to view the results of the analysis conducted by an external tool which identifies a measure of technical debt and individual code smells in their source code.

## **2.6 Non-Functional Requirements**

### Performance

The application is developed as a prototype for this project. Therefore, it is hosted locally on a single machine and is not required to handle simultaneous users. However, the application should employ a robust server-side framework and should be readily deployable to a hosting service where it could handle simultaneous users.

As the prototype application is hosted locally, a more efficient database can be used to maximise the speed of data storage and retrieval. SQLite should be used to avoid unnecessary and time-consuming requests to a hosted database. However, the application should be designed such that a hosted database could be easily connected if the decision was made to deploy the

application. A suitable ORM tool should be used so that a hosted database can be connected without changing any of the existing functionality.

### User Interface

The user interface should be clear and user-friendly. A suitable UI framework should be used for consistency and professionalism. A navigation bar should be present on every page containing quick-links to other areas of the application, consistent with many websites. The colour scheme should be consistent across the application and should be used to highlight important information. Graphs should be rendered such that they can be interpreted intuitively.

### User Experience

System response time should be less than 2 seconds for all pages that do not require data to be fetched from external sources. For pages which do fetch external data, such as routes which require an API call, response times should be less than 5 seconds, however this is dependent on the strength of the internet connection.

The most time-intensive processes; such as downloading the source code from numerous repositories and analysing it for quality; should be performed externally to the main application. This has the effect of allowing the user to continue using the application without having to wait on long processes finishing.

Some pages require intensive calculations to be performed, such as combining quality scores for user-chosen projects. As the response requires the results of calculations to be known before any page can be rendered, this wait is unavoidable unless the application limits the number of repositories that can be compared simultaneously. However, this would diminish the value of the application and should not be implemented.

The application requires rendering of various graphs which can be time-consuming. Any pages which require graph-rendering should load within 5 seconds.

Steps should be taken to make the application as user-friendly as possible. Pages which suggest repositories that the user may wish to analyse should allow the user to copy the repository URL to the clipboard with a single click. AJAX should be used where possible to make asynchronous requests without disrupting the user experience by reloading the current page. AJAX could be used for filtering repositories by language, for example. Finally, a cookie should be set upon login so that the user is not prompted to log in each time they use the application.

## Reliability

The system should not crash unexpectedly. All application links should be existent. Any user input that causes a component to fail should be reported clearly to the user, and the application should redirect the user to the requesting page. The system should have 100% server-side unit test coverage.

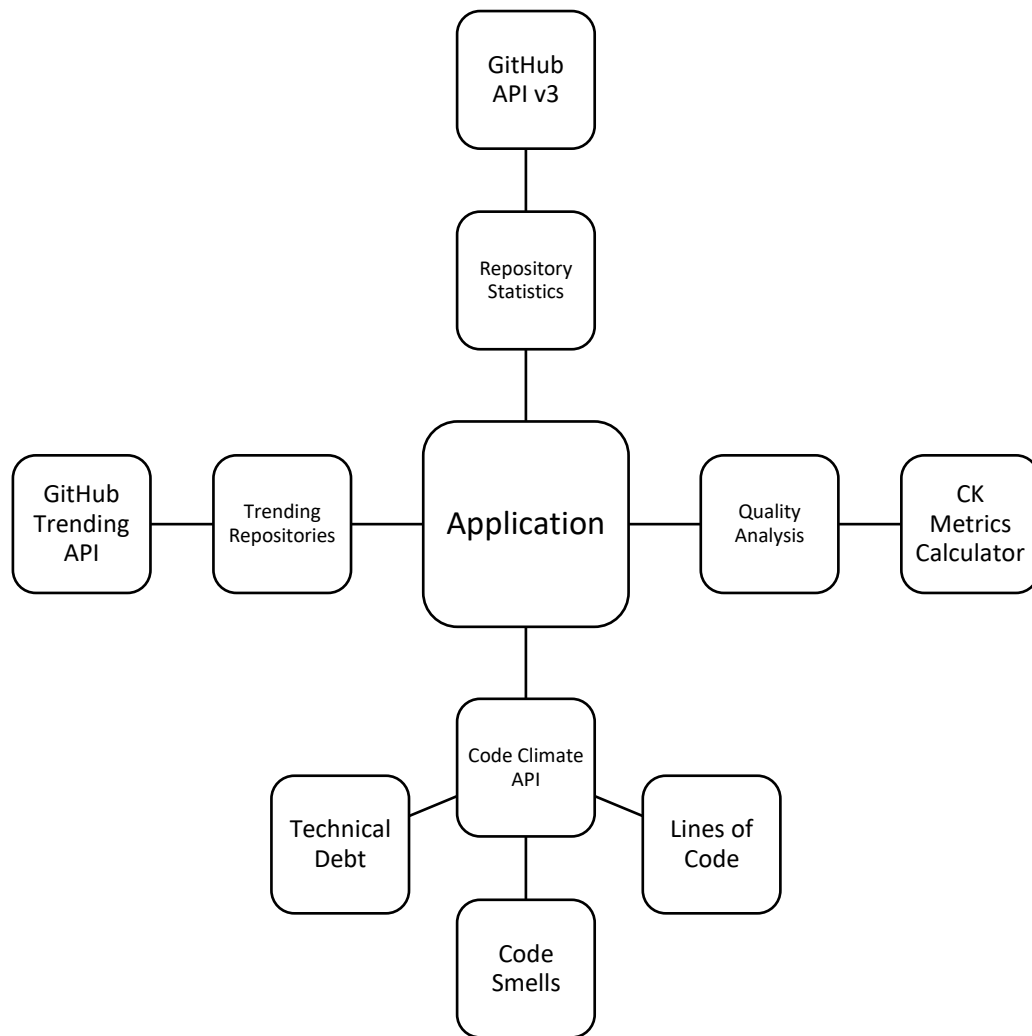
## **2.7 Interfaces**

This application will utilise a number of interfaces to provide data necessary to this project's investigations. A diagrammatic interpretation of the interfaces used is shown in **FIGURE 6**. The main two interfaces are the GitHub API v3 and the CK Metrics Calculator. The GitHub Trending API and Code Climate API provide additional data for popular repositories and code smells.

The GitHub API has multiple uses. Firstly, the API provides a method of downloading copies of open-source (public) repositories so that quality analysis can be performed on the source code. Additionally, the GitHub API can be used to source the data required for the investigations implemented in this project. Data describing the commits made to a repository between specified time intervals can be extracted, and from this, we can calculate the code change frequency and the number of contributors to a project.

The CK Metrics Calculator is an executable Java file which analyses source code in a specified directory according to the CK Metrics. This tool provides the individual metrics which are used to calculate code quality. The tool is open-source and can be found on GitHub [18]. Upon initial execution of the tool, some errors were detected, and the code was modified to resolve these. Some of the functionality was also removed as it was unrelated to this project in order to improve the execution time of the tool.

**FIGURE 6: Application Interfaces**



## 2.8 User Characteristics

Two types of end users are intended to use the application: researchers and software developers. Each user type will interact with the application differently, and the requirements are therefore different for each.

### Software Developers

In this context, software developers are considered to be anyone who is involved in the software development process, including those in non-technical or managerial roles who are responsible for coordinating development. These users will use the application to analyse the quality of their code in order to make decisions about actions to take to improve their development process.

The application should allow users to provide their code via a link to their GitHub repository and view the results of quality analysis. In addition to providing a quantitative measure of code quality, the application should provide users with an indication of the areas of their code which are negatively affecting quality, including measurement of technical debt and detailed information about code smells. This information is intended to direct users to areas of their code which could be improved to increase the quality of their code, and users can analyse their codebase over various iterations to determine whether specific development choices or refactoring efforts have been effective. Quality comparison between a software product and its competitors is another possible use of this application, provided that both products are open-source.

### Researchers

The application is also intended to be used as a tool for researching the various factors that affect code quality. Therefore, the application should provide a method of selecting GitHub projects which satisfy specified criteria defined by the researcher. The user should then be able to submit the selected projects for analysis over a defined time period and view the results as clearly-formatted visualisations. The application should perform statistical analysis on the results to provide context and validity to any trends shown by visualisations.



## 3.0 Design

### 3.1 Application Architecture

The application will be developed using the Flask framework for Python. Flask is a microframework which is based on Werkzeug and Jinja2. Werkzeug provides the WSGI interface while Jinja provides the templating. However, Werkzeug is not recommended for production servers so a Gunicorn server will be used to host the application. Gunicorn is an alternative WSGI server implementation which is more stable than Werkzeug. This application is not intended to be deployed to a production environment; however, the development approach is to design an application that could be easily deployed if required.

Flask can be structured to follow the Model-View-Controller design pattern. Each component of MVC in relation to this application is detailed below:

**Model:** This component handles all interactions with the application data. The Flask-SQLAlchemy extension provides an ORM for SQL databases and will be used in conjunction with SQLite3 to handle and store the application data.

**View:** This component is responsible for the user interface. The view is updated by the controller to provide the user interface and conveys user interaction back to the controller. Jinja2 will be used to provide dynamic templating along with HTML, JS and CSS for static templates.

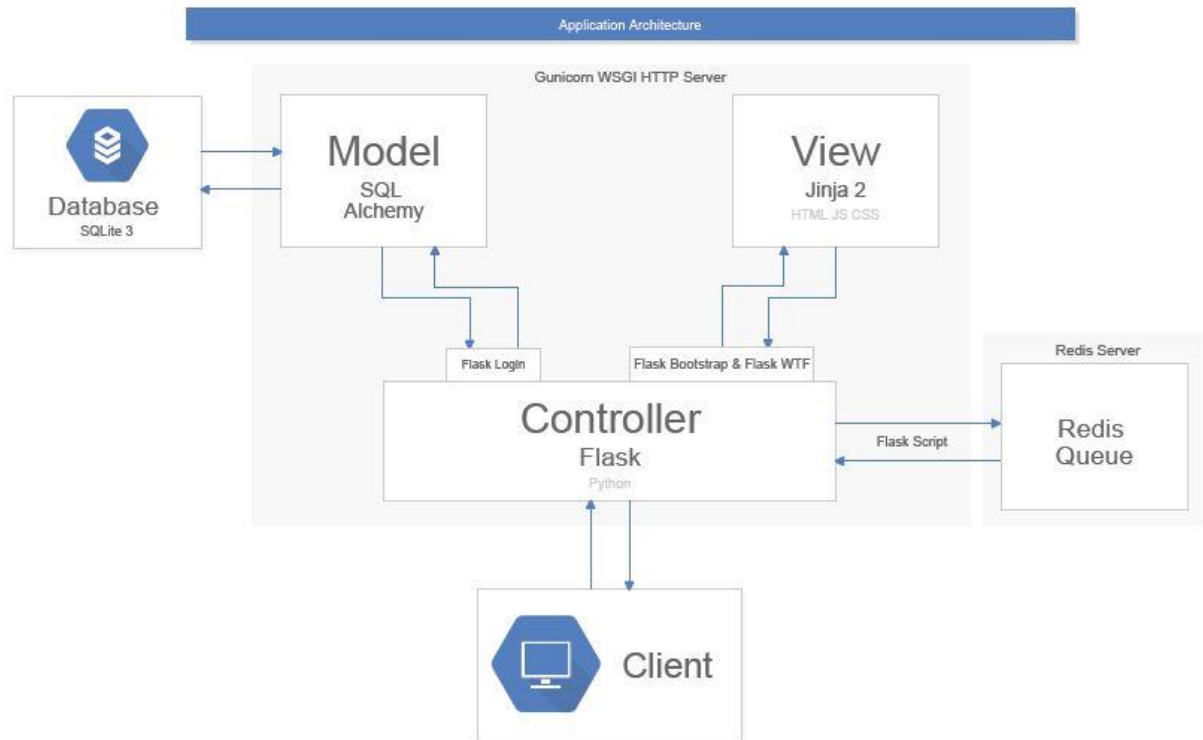
**Controller:** This component is responsible for organising interaction between the view and the model. It orders the model to update its state and orders the view to update the user interface. Python will be used with Flask to handle routing, initialisation and execution.

Flask extensions such as Flask-Bootstrap, Flask-Login and Flask-Script will be used to provide the additional functionality that is required but is unavailable from Flask out-of-the-box as it is a microframework.

In addition to the main application, a second process will be required to execute the downloading and quality analysis of projects. These processes are time-consuming, and if they were run in the main application context, the application would be unusable for long periods while source code is downloaded and analysed, negatively impacting the user's experience. Therefore, a separate Redis server instance will run alongside the application and the Python

library RQ will be used to implement a job-queue that will execute time-intensive jobs separately from the main application. The system architecture is displayed in **FIGURE 7**.

**FIGURE 7: Application Architecture**



### 3.2 User Interface Design

The application requires a user interface for discovering GitHub projects, analysing them and viewing the results. The user interface will have the form of a web application. The non-functional requirements state that the user interface should be simplistic and professional. Therefore, emphasis will be placed on delivering an interface that focuses on ease-of-use and clarity.

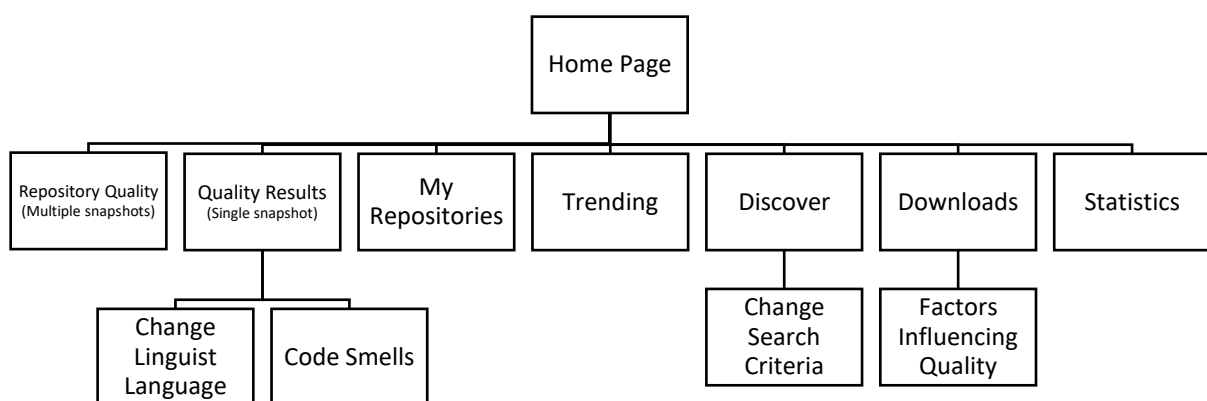
For the user interface to appear professional, a suitable framework will be utilised throughout all pages of the application. Twitter Bootstrap 4 is a widely-used open-source framework that can be used to create websites with minimal design effort. Bootstrap is also a responsive framework, meaning that it scales automatically depending on the size of the device's viewport. These qualities make Bootstrap a suitable choice for this project, as the application is intended for the use of its functionality rather than having a bespoke interface. However, the user interface should still be appealing, which can also be accomplished using Bootstrap. The final relevant quality of Bootstrap is its minimalistic-styling. Minimal styling is suitable for this application as user focus should be directed towards the content of web pages rather than their appearance.

Navigating the user interface will be straightforward and consistent. The three main sections of the application: discover repositories, perform analysis and view results; will be quickly accessible from any page via the navigation bar. The navigation tree should be shallow, meaning that the user will not be required to click through many different pages to access any functionality or information. **FIGURE 8** illustrates the sitemap.

The application will require minimal effort from the user to complete actions. User input will be kept to a minimum, and all forms will be validated with informative error messages. User interface widgets, such as date pickers, will be used to simplify user input and perform some initial validation. For pages which list repositories that the user may wish to analyse, shortcut buttons will be present that copy the URLs of user-selected repositories to the clipboard, so that they can be quickly and easily submitted for analysis. The multiple processes of downloading the source code, analysing its quality and extracting GitHub statistics will be executed with a single click from the user.

Some other notable user interface implementation details include the use of AJAX to improve the user experience on pages which involve searching, having a consistent colour scheme across the application which highlights important information, and rendering any visualisations such as graphs with appropriate formatting so that they can be interpreted intuitively. Some of the user interface design considerations regarding delivering a high-quality user experience are included in Appendix 1.

**FIGURE 8: Sitemap**



The application *home page* will consist of a form which allows the user to archive one or more repositories and a list of the repositories that have already been archived. Archiving a repository refers to downloading and analysing the source code of one or more versions of the repository over a defined time period and extracting the necessary information from the GitHub

API, as defined in section 2.2 of this document. The user will be able to change the method of input by clicking a button on the form which provides a more convenient interface for archiving multiple repositories simultaneously.

The “*My Repositories*” and “*Trending*” pages list repositories that are owned by the current user and repositories that are defined as popular by GitHub respectively. The GitHub Trending API provides the source of data for the “*Trending*” page by measuring user interaction with public repositories in the last day, week and month. The user will be able to filter the trending repositories by programming language using a background AJAX request. Both pages will provide shortcut buttons which allow the user to copy the URL of any of the repositories to the clipboard for use on the home page.

The “*Statistics*” page consists of a form which is similar to the form on the application home page and allows the user to submit one or more GitHub repository URLs in order to view statistics for the repositories. Submitting the form sends a request to the GitHub API for information about the repositories. The information requested for each repository includes the number of stargazers, subscribers, watchers, open issues, forks and branches. The repository size in bytes is also retrieved from GitHub. The data will be formatted as an interactive bar chart using the Google chart jQuery plugin, and the user will have the option to switch to the raw data view.

The “*Discover*” page displays the results of a search query using GitHub’s API. The default search finds public Java projects between 1Mb and 5Mb in size, created on GitHub at least three years before the search date and updated within the last 30 days. The page will contain a link to a form which allows the user to modify the search criteria. The user will be able to select one or more repositories and copy the URLs in a format required for the form on the home page.

The “*Downloads*” page is used as a redirect location when the user wishes to compare repositories. The page will display a list of all repositories that the user has previously archived and allow the user to select some or all of the repositories for comparison. Once repositories have been selected, the user is directed to the “*Compare*” page where the comparisons between projects are visualised. The Compare page contains graphs which illustrate the change in code quality as a result of time, code size, contributors and change-frequency. This page facilitates the investigations described in section 2.2 of this document.

As mentioned previously, the home page lists all of the repositories that have been archived. When comparing repositories against each other, the user will use the “*Factors Influencing Quality*” route. However, the application also permits the user to view more

information about the quality score for a repository or to compare a repository to previous versions of itself. This functionality is available via links on the home page.

The “*Quality Results*” page is where users are routed after clicking on a single version of a repository from the home page. This page displays the quality score for the specific version of the repository, as well as scores for each of the CK Metrics which contribute to the overall quality score. The user is also provided with information about the repository that is collected via Code Climate. This information details technical debt and code smells identified by Code Climate. A link will be available which forwards the user to the “*Code Smells*” page where they can discover more detail. The language that has been analysed will be visible at the top of the page. The application will use the primary language as defined by GitHub’s Linguist Language tool when performing analysis. However, the Linguist Language library does not always guess correctly, so the “*Quality Results*” page will provide a link to a form where the user can set the primary language of the repository in order to receive a more accurate analysis.

The “*Repository Quality*” page is the alternative to comparing a single version of a repository. This page is accessible via a link on the home page which allows the user to analyse a single repository over multiple iterations. This page contains visualisations detailing the quality of the repository’s source code over each of the versions that have been archived, in addition to some of the statistics extracted from the GitHub and Code Climate APIs, such as quality vs technical debt and quality vs the number of issues.

The “*Code Smells*” page contains information extracted from the Code Climate API which helps the user to identify areas of their code which should be improved. These quality issues are often referred to as code smells. Code Climate identifies code smells on a per-class basis, and all classes that contain code smells are listed on the page with more detail regarding the type of issue and the specific lines of code. The information will be summarised at the top of the page for the user’s convenience. Code Climate also assigns each repository a “grade” ranging from A-F based on code quality. The grade will be displayed alongside the summary information.

Wireframes have been drawn to illustrate the user interface design for each web page. These are included in Appendix 2.

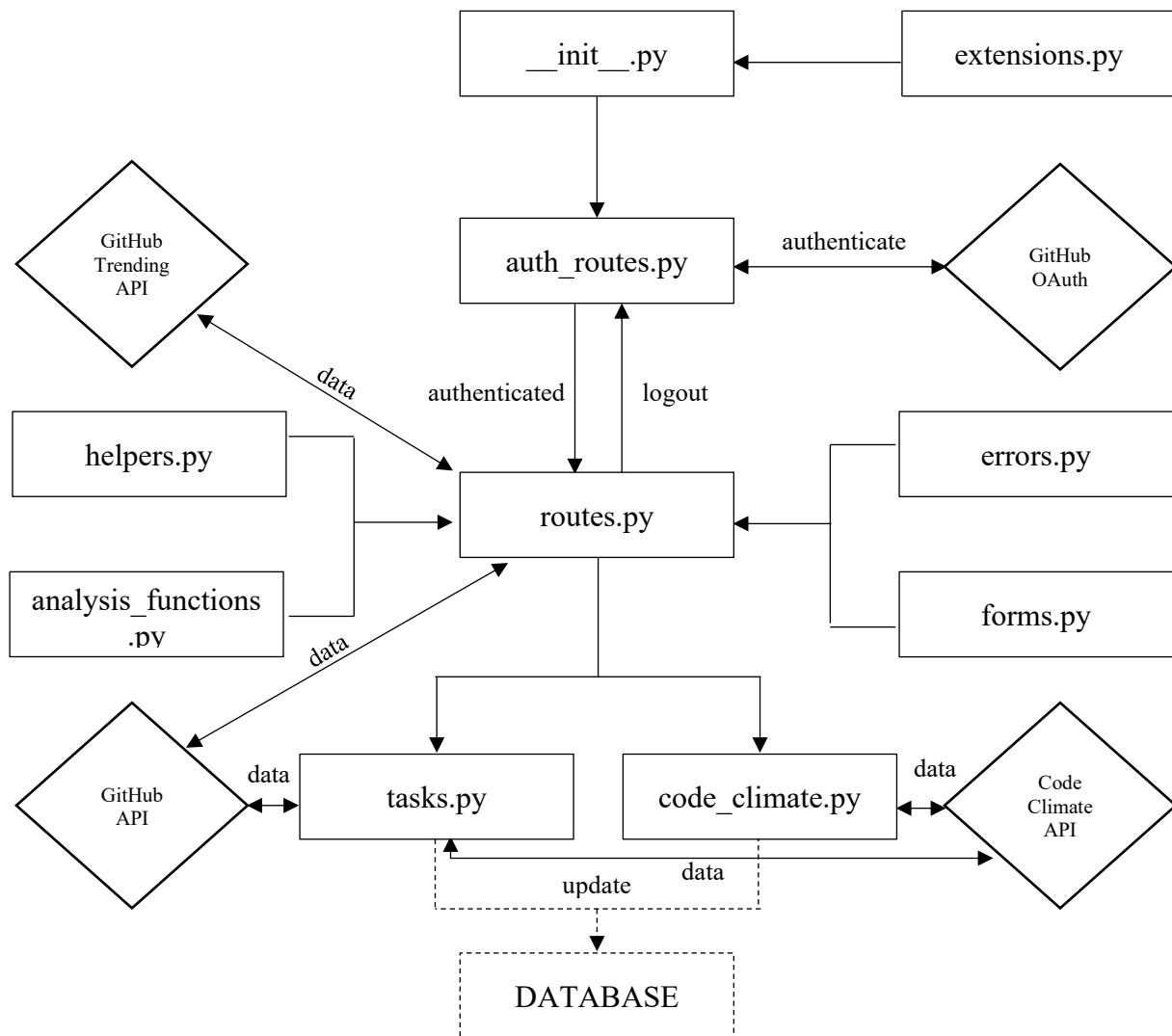
### 3.3 Software System Design

This section describes the application components at a high level. A component is an individual Python module. Logical operations involving distinct elements of the application will be separated into individual components in order to maintain a good application structure. **TABLE 4** describes each component. Implementation details can be found in section 4 of this document.

**TABLE 4: Software System Design**

Component	Description	Interfaces
__init__.py	Initialises the application. Creates defined blueprints for individual components, imports extensions.	
extensions.py	Initialises the Flask extensions used in the application.	
models.py	Defines each data model used in the application. Provides a reference for the ORM.	
config.py	Defines individual classes containing configuration options for the application, e.g. Production, test config.	
constants.py	Stores constant values that need to be referenced by components of the application, e.g. API header values.	
tasks.py	Defines the functions that are accessible to the external Redis server for execution outside of the main application. Defines the functions used to download source code, calculate its quality and extract information from the GitHub API.	GitHub API, Code Climate API
calculate_ck_metrics.jar	Executable Java program used to calculate CK Metric values for provided source code.	
routes.py	Defines all of the main application routes.	GitHub API, GitHub Trending API
auth_routes.py	Defines all of the OAuth authorisation routes.	GitHub OAuth
errors.py	Defines routes to handle HTTP errors, e.g. 401, 500.	
forms.py	Defines forms and custom validation methods used for user input.	
jinja_templates.py	Defines custom helper templates for Jinja rendering.	
analysis_functions.py	Defines functions that are used to correlate code quality with time, code size, change-frequency and number of contributors, and perform statistical analysis.	
code_climate.py	Contains functionality required to push repositories to Code Climate for analysis and retrieve the results from the Code Climate API.	Code Climate API
data_dict.py	Defines some data required by the application which can be configured by a developer, e.g. CK Metric thresholds	
helpers.py	Defines helper functions used to aid the developer, e.g. processing paginated API responses, formatting data for chart rendering.	GitHub API

**FIGURE 9: Component Interactions**



### 3.4 Database

A data store is required for the application data. The application will utilise an SQLite3 database which will be hosted locally. The database will store all of the information for registered users and archived repositories. A total of seven tables will be required to facilitate the storage of the application data. The schema is illustrated in **FIGURE 10**.

The *user* and *flask\_dance\_oauth* tables store all of the data relating to individual users of the application. When the user first accesses the application, they are redirected to GitHub's OAuth portal where they can log in using their existing GitHub credentials. This means that this application is required to store very little information about the user. Flask Dance helps orchestrate the OAuth login and stores the GitHub access token generated by the authentication process in the *flask\_dance\_oauth* table. The *user* table simply stores a unique identification number and the user's GitHub username.

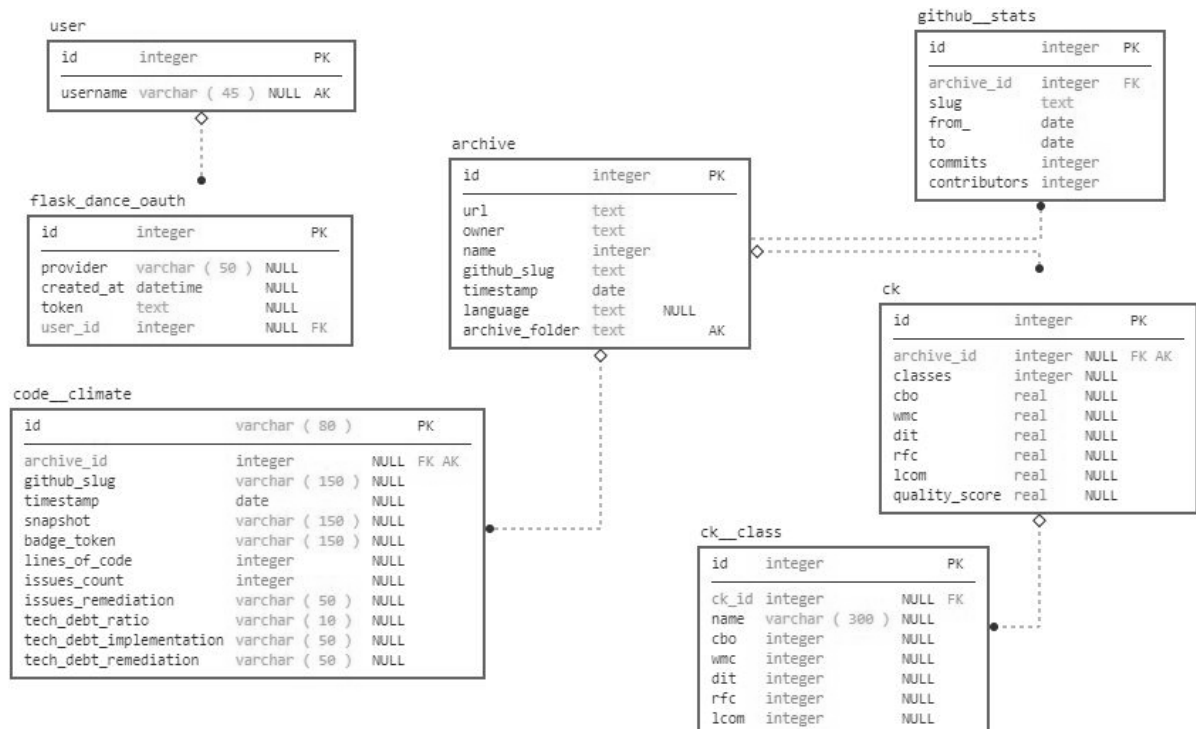
The *archive* table stores information about every version of a repository that is archived by the user. The GitHub URL and slug are stored along with the relative path to the location of the downloaded source code and the date to which the version of the repository correlates.

The *code\_climate* table stores all of the information relating to each version of a repository that has been committed to code climate for analysis. The Code Climate repository identification data is stored in addition to the results of Code Climate analysis, such as technical debt ratio and lines of code.

The *ck* and *ck\_class* tables store the results of code quality analysis for each version of each repository. The *ck\_class* table stores the individual metric scores for every class in each repository. The *ck* table stores the total number of classes, general metric scores and the overall quality score for each repository.

The *github\_stats* table stores all of the information extracted from the GitHub API about each repository. The dates between which the data has been taken are stored in addition to the average monthly number of commits and unique contributors for each repository between the two dates.

**FIGURE 10: Database Schema**





### **3.5 Concurrency**

The process of downloading the source code for a repository, submitting it to Code Climate, analysing its quality and extracting information from the GitHub API will be time-consuming. Even for small and medium-sized repositories, although the process should be relatively fast, the application would be unusable while the process is executed. Therefore, it would be impractical to make the user wait for this process to finish. A separate execution thread will be used to resolve this problem. This will enable the archive process and main application to run concurrently, allowing the user to continue using the application while the archive job runs in the background. A Redis server instance will be used to host an individual worker which will facilitate the execution of archive jobs in the background, without impeding the main application. The Flask extension Flask-RQ implements a job queue which will be used to execute the individual functions of the archive process in chronological order.

### **3.6 Fault Tolerance and Error Handling**

Application errors will be handled when they occur and will be reported to the user using Flask's built-in Flash messages. HTTP errors may occur outside of the application's control, but these will also be handled using flask error handlers to prevent the application from crashing completely. For example, if the user terminates a request early while the application is reading incoming data.

One exception to the handle-when-encountered rule is the occurrence of an exception during any jobs executing on the Redis server, as these will not cause the main application to crash. Flask-RQ handles exceptions internally by moving them to a separate "failed" queue. Therefore, exceptions encountered when archiving a repository can be logged, and the process can continue. Exceptions may occur due to a repository being non-existent during the user-specified time period, for example. Rather than aborting the entire job, the exception will be logged, and the remaining repositories will be archived as normal. The user will be notified of the number of failed repository archive attempts.

## 4.0 Implementation

### 4.1 Implementation Decisions

The application is web-based using the Flask framework with Python. Flask was chosen due to its lightweight design which minimises design complexity and helps limit the application size. Flask has a large number of well-documented extensions which provide the functionality that the framework does not provide as standard. Extensions can be added as required, reducing the amount of unused code in the application.

Flask is based on the Python programming language which is useful for both web-application server-side development and data analytics and is therefore suitable for this project. Python is a useful language for rapid application development and offers numerous helpful and robust libraries.

### 4.2 Software Libraries

External software libraries have been used to implement some functionality where possible. Existing Flask extensions are easily integrated into the application and are usually well-tested and robust. Some existing Python libraries are also used. **TABLE 5** outlines the external software used in the application.

**TABLE 5: Software Libraries**

<b>Library</b>	<b>Description</b>
Flask-Login	Provides user session management and handles login and logout as well as storing users' sessions so that the user is not required to log in each time they use the application.
Flask-Dance	Handles OAuth integration with GitHub, allowing the user to log into the application using their existing GitHub credentials.
Flask-SQLAlchemy	An object-relational mapper (ORM) that handles all interactions with the database using developer-friendly objects mapped to database tables.
Flask-Bootstrap	Provides the base template for HTML web pages that extends the Twitter Bootstrap framework, allowing web pages to be developed quickly.
Flask-WTF	Provides a simple interface with the WTForms library. WTForms can be defined in Python and rendered easily within HTML. Also provides basic form validation which can be extended.
Flask-Script	Provides support for writing external scripts in Flask. Used with Redis server and Flask-RQ to enable some tasks to be executed separately from the main application thread.

Flask-RQ	Provides a method of adding tasks to an external job queue where they can be executed on the external Redis server. Handles job execution including error handling and exceptions.
Pygit2	A set of Python binding to the libgit2 library. Libgit2 is a C implementation of the core Git methods. Provides a wrapper for executing Git commands within Python code.
PyGitHub	A Python library for accessing the GitHub v3 API. Helps with management of GitHub repositories and user profiles.
Git	Provides object model access to Git repositories. Used to perform Git operations in Python code.

In addition to the software libraries used for backend functionality, a number of libraries are used for the application front-end. Twitter Bootstrap provides the base template for each of the web pages as well as some of the user interface JavaScript functions that facilitate user input on WTForms, for example. Google Charts are also used to render the graphs used in the application.

### 4.3 Component Implementation

This section describes the methodology for some of the logical components of the application. The components listed below are responsible for downloading single or multiple versions of repositories, analysing their quality, uploading them to Code Climate and obtaining information about them from the GitHub API.

#### Downloading Source Code

Downloading the source code of a repository is the first step in the analysis process. Access to a local copy of the source code is required by the program that is responsible for calculating the CK Metric values for each of its classes. In order to download the source code for a repository, the user submits the archive form on the application home page. When doing so, the user enters one or more GitHub repository URLs and chooses a period of time over which to analyse the repositories using the date picker widgets. The user can choose two separate dates to analyse different versions of the repositories between the selected dates, or they can choose a single date on which to analyse a snapshot of each repository. A snapshot is an exact representation of the repository as it was on the exact date selected.

In order to download a single snapshot of a repository, the application begins by downloading the current version of the repository from GitHub to the applications archive directory using the Pygit2 module. The Git module is then used to initialise a Git repository in the download directory. If the chosen date is not today's date, the application accesses the

commit history for the repository using the GitHub API and obtains the SHA-1 hash of the most recent commit on the chosen date. This identifier allows the application to revert the newly initialised Git repository to a past version of itself which is a replica of the repository on the selected date.

If the user has selected two different dates over which they wish to analyse the repository, a function is used to determine the time intervals over which to download versions of the repository. If the chosen dates are less than five days apart, the application downloads a version of the repository on each of the dates chosen using the method described above. If the two dates are more than five days apart, a calculation is performed which splits the amount of time between the two dates evenly into five intervals. The first interval is the first chosen date, and the last interval is the second chosen date. The remaining intervals are evenly-spaced between these two dates. A version of the repository is downloaded on the date of each interval using the method above.

The processes above are repeated for each of the repository URLs submitted by the user on the RQ worker that operates separately from the main application. In cases where the repository did not exist on one of the calculated time intervals, the source code cannot be downloaded. However, this error is caught and logged, and the process is allowed to continue for other versions of the repository or repositories. This error handling is required as the user may wish to analyse multiple versions of multiple repositories over a period of time during which one or more of the repositories may not have existed. Rather than limiting the user to archiving versions of the repositories that existed simultaneously, this method downloads as many versions as are possible and reports any errors to the user later.

### **Analysing Code Quality**

Once the source code has been downloaded for a repository, quality analysis can begin. Mauricio Aniche developed an open-source tool which can be used to calculate each of the metrics in the CK suite for provided source code on a per-class basis [18]. This tool was written in Java and can be compiled into an executable that can be executed by the application using Python's subprocess module. The tool was modified for use in this application in order to resolve some bugs and remove some redundant functionality to improve execution time.

The JAR file is executed in the directory containing the source code for each version of each repository that is being archived. The class-level metric scores are written to the database before the overall metric scores are calculated using the formula in **FIGURE 1**. The repository

quality score is then calculated for each version of each repository using the formula in **FIGURE 3**, and the results are written to the database.

### **Code Climate**

The Code Climate tool [17] provides information about code smells and technical debt which is used by the application to direct users towards areas of their code which could be refactored to improve the code quality. Code Climate also provides a count of the total number of lines of code in a repository. In order to submit a repository to Code Climate for analysis, the source code must be added to the site, similar to how source code is pushed to GitHub. However, the user pushing the source code must be the owner of the repository, meaning that repositories owned by GitHub users other than the current user cannot be added to Code Climate directly, despite them being open-source. A placeholder user account was created on GitHub, and the already-downloaded source code is pushed to the placeholder account to resolve this issue. The dummy user is now the owner of the repository and can submit the code to Code Climate on behalf of the current user via the Code Climate API.

### **Extracting GitHub Statistics**

The final step in the analysis process is to extract the necessary data from GitHub for each version of each repository using the GitHub API. Specifically, the application needs to count the number of commits and the number of contributors for each repository. This information is used to compare the effect on code change-frequency and number of developers against code quality. Code change-frequency is measured as the average number of monthly commits made to a repository. The number of contributors is measured by counting the number of unique users who made these commits. This data is calculated by extracting the commit history for each repository between the start and end dates and processing the data to obtain the required values. The calculated values are written to the database.

## 5.0 Testing

The application will be tested using unit tests and acceptance tests. Unit tests will focus on the implementation of functionality, ensuring that the application outputs expected results, handles exceptions correctly and does not crash unexpectedly. Acceptance testing will ensure that the user can interact with the application as intended and that the application allows the user to achieve the use cases defined in section 2.5 of this document.

### 5.1 Unit Testing

The objective of unit testing is to validate the application's correctness. Unit tests have been written for use at the source code level to discover flawed-logic and syntax errors. The unit testing strategy for this application uses two frameworks: one to test functions that use the Flask framework, such as routing functions, and one to test functions written purely in Python, such as the function responsible for calculating code quality. The two frameworks used are Flask-Testing and unittest. Unit tests for this application are written as automated tests which means that all tests can be executed together by running the test class. A mock database is created by the test setup methods so that the application database is unaffected by test data.

The unittest framework is based on Java's JUnit framework and supports automated tests with shared setup and teardown code for tests. The shared setup and teardown code means that necessary objects can be initialised before each test is run and deleted after completion of the test. The unittest framework is used to test the pure-Python functions within the application, for example, testing the helper module.

The Flask-Testing framework is used to test the application routes and server functions that are implemented by Flask, such as login. **TABLE 6** outlines the modules that have been tested and the code coverage for each module. Unit testing outcomes and coverage evidence are included in Appendix 3 and Appendix 4.

**TABLE 6: Unit Test Coverage**

Module	Number of Methods	Number of Tests	Pass	Fail	Test Coverage
(auth)routes.py	12	28	100%	0%	98%
code_climate.py	7	24	100%	0%	100%
helpers.py	13	36	100%	0%	100%
jinja_templates.py	3	15	100%	0%	100%
tasks.py	11	24	100%	0%	100%

## 5.2 Acceptance Testing

Acceptance testing is used to guarantee that the system meets its specified requirements. The requirements for this application are defined as use cases in section 2.5 of this document. **TABLE 7** lists each use case with the method of testing and evidence that the application meets the requirement. Additional acceptance tests can be found in Appendix 19.

**TABLE 7: Acceptance Tests**

Use Case	Requirement	Test Scenario	Pass Evidence
1	Successful User login with valid credentials	Navigate to the application home page as an unauthenticated user. Sign into the application using GitHub credentials.	Appendix 5
2	Visualise GitHub statistics for different projects	Submit the form on the statistics page with 2 GitHub repository URLs. View the graph and raw data.	Appendix 6
3	View trending repositories	Navigate to the trending page. View trending repositories. Filter for Java repositories today.	Appendix 7
4	Access quick-links to user repositories	Navigate to the my repositories page. Use the buttons provided to copy a repository's URL to the clipboard.	Appendix 8
5	Find GitHub repositories for analysis	Navigate to the discover route. Observe repositories found using the default search criteria	Appendix 9
6	Find GitHub repositories for analysis using custom search criteria	Navigate to the discover route. Click on the change criteria link. Change the value for each filter (language: "C", min size: 5000, max size: 10000, created before: 01/01/2016, updated after: 01/04/2019). Observe repositories found using the updated search criteria	Appendix 10
7	Analyse Source Code for a single version of one repository	Paste a single repository URL into the form on the home page and select the same date for start and end date.	Appendix 11
8	Analyse Source Code for a single version of multiple repositories	Paste 3 repository URLs into the form on the home page and select the same date for start and end date.	Appendix 12
9	Analyse Source Code for multiple versions of a single repository	Paste a single repository URL into the form on the home page and select different start and end dates.	Appendix 13
10	Analyse Source Code for multiple versions of	Paste 3 repository URLs into the form on the home page and select different start and end dates.	Appendix 14

	multiple repositories		
11	Visualise code quality and GitHub statistics for multiple versions of a single project	Navigate to the application home page. Find a repository with multiple versions and click the compare <repository name> repositories link. Observe the graphs produced and ensure all versions of the repository are represented.	Appendix 15
12	Visualise code quality and the factors affecting it for multiple versions of multiple projects	Navigate to the compare route. Select three repositories and click “compare selected”. View graphs produced and ensure all three repositories are represented.	Appendix 16
13	Change the default language of a repository	Navigate to the change language route. Observe the current language for the repository. Use the form to change the language to an alternative. Navigate back to the change language route, ensuring that the language has been changed successfully.	Appendix 17
14	View specific issues that are reducing code quality	From the home page, click on a repository to analyse it. Scroll to the bottom of the results page and click the view issues link. Ensure that issues for the project are listed and expand the accordion to view issues in different files.	Appendix 18



## 6.0 System Evaluation and Experimental Results

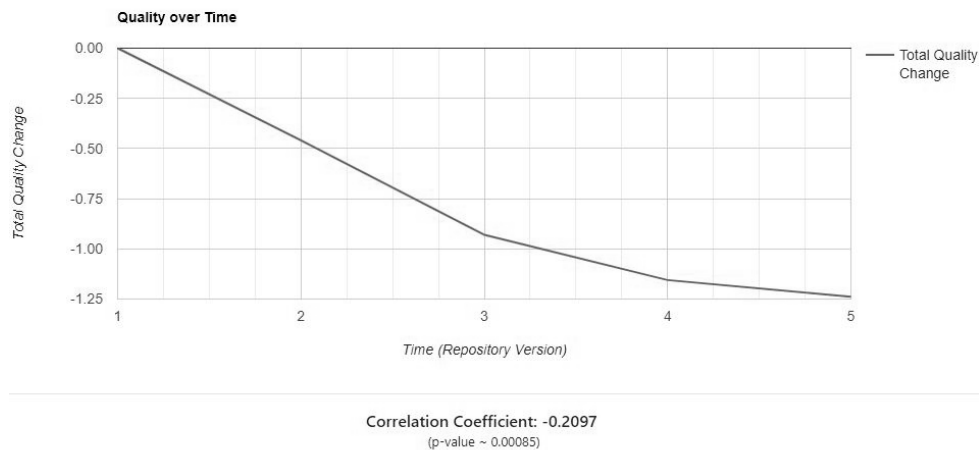
The objective of this project was to investigate how various developmental factors influence code quality. The research involved four factors: time, code size, change-frequency and number of contributors, each with a single hypothesis. A web application was developed which facilitated the investigation of each of the hypotheses and provided additional functionality for further use by helping developers understand more about the quality of their code. The results from the investigation of each hypothesis help to answer each research question.

The methodology was consistent for each investigation. Fifty randomly-selected GitHub repositories were chosen from the discover page of the application, which listed public Java repositories that were between 1Mb and 5Mb in size, a minimum of 3-years-old and had been updated within the last 30 days. Five versions of each repository, evenly-distributed over a 3-year period, were download and quality was calculated for each version.

### 6.1 Code Quality over Time

H1 proposed that code quality generally decreased over time. This hypothesis was investigated by analysing the change in the code quality of each of the 50 repositories starting from the earliest version and finishing with the most recent. The aggregate change in quality across all repositories was calculated for each of the five intervals by comparing the total quality change from the previous interval to the current interval. The total quality change was set to 0 for the first interval. The values were plotted on a line graph, and the result appears to verify the hypothesis. Code quality was observed to decrease consistently with each iteration of the repository version — however, the magnitude of the loss of quality was small. The average loss in quality was approximately 6.7%, from an average initial quality of ~0.368 to average quality of ~0.343 at the end of the 3-year period. Spearman's Rank correlation coefficient was used to validate the apparent trend shown in **FIGURE 11**. The resultant correlation produced was approximately -0.21, indicating a “weak” negative correlation. However, the calculated p-value was <0.0009 suggesting that the probability of this correlation happening by chance is extremely low. Considering that confidence-level 95% ( $p=0.05$ ) is widely accepted as statistically significant, we can infer that the hypothesis can be accepted that code quality decreases over time, albeit trivially.

**FIGURE 11: Quality over Time**



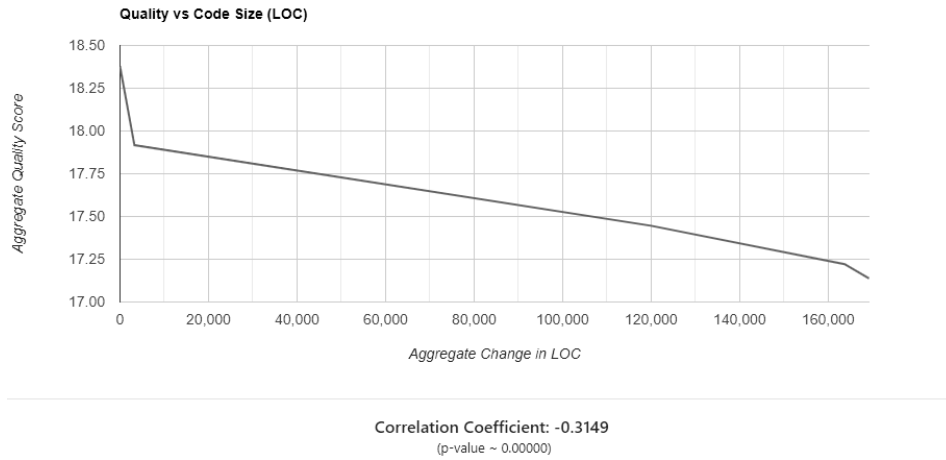
## 6.2 Code Quality vs Code Size

H2 suggested that code quality decreases with increased code size. This hypothesis was investigated by comparing the total code quality of all repositories against the total accumulative change in code size for each repository across each of its versions, i.e. the sum of the quality scores of all repositories for each version compared with the sum of the number of lines of code added between consecutive versions. The aggregate change in code size across all repositories was calculated for each of the five versions by comparing the total number of lines of code from the first version to the current version. The total code size was set to 0 for the first version. The change in code quality was calculated using the same method, substituting the number of lines of code for code quality.

The results were plotted on a line graph as shown in **FIGURE 12**. The graph shows a negative relationship between code quality and code size. The graph shows a sharp loss in code quality between the first and second versions, despite a comparatively small change in total code size. However, the magnitude of the loss in quality is consistent with the loss in quality with significant increases in code size. This is a possible indication that the magnitude of the increase in code size is less relevant than the mere fact that code size increased. Further study could be undertaken to investigate this more closely.

The Spearman's Rank correlation coefficient was approximately -0.31, with a p-value of  $3.7 \times 10^{-7}$ , indicating a "weak" negative correlation with high statistical significance. Therefore, H2 can be accepted that code quality is negatively affected by increasing code size. This result supplements previous work by Jay et al. who, using a different measure of code quality for C++ projects, found that code size and complexity – a contributor to code quality for this study - have a near-perfect linear relationship [19].

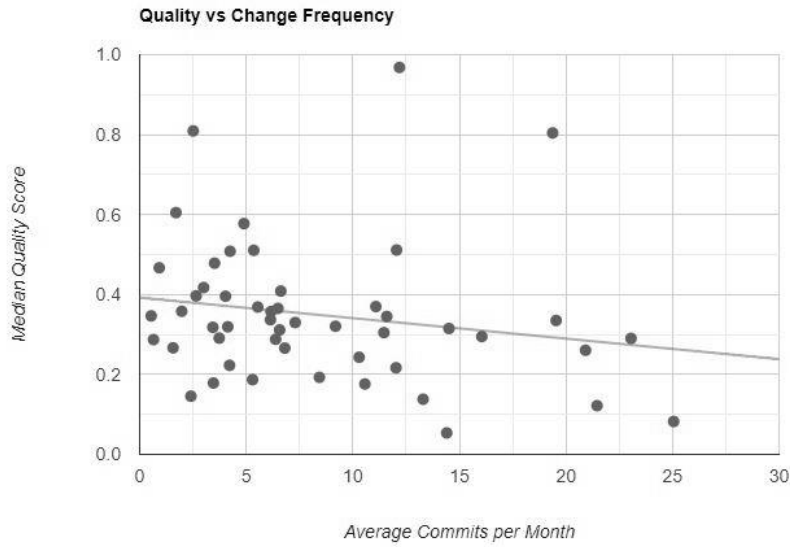
**FIGURE 12: Quality vs Code Size**



### 6.3 Code Quality vs Change-Frequency

H3 proposed that code quality is higher in projects that are changed frequently. Change-frequency was measured as the average number of commits per month over the 3-year period for each repository. The median code quality was then computed for each repository over the same period, and the values were plotted on a scatter graph which is shown in **FIGURE 13**. The median was used as the quantifier of average code quality to negate the possible imbalance caused by projects that were created at the beginning of the 3-year period, and which may have had little code from which to derive an accurate quality score. The line of best fit on **FIGURE 13** suggests a negative linear relationship between the two attributes which is contrary to our hypothesis. Upon statistical analysis of the data, the Spearman's Rank coefficient was approximately -0.29 which does indicate a "weak" negative correlation. The computed p-value, 0.044, although below the alpha value 0.05 and therefore statistically significant, is notably larger than the p-values recorded for H1 and H2, suggesting that the correlation is more vulnerable to random occurrence. **FIGURE 13** also shows some contradictory values, for example, a quality score of ~0.8 can be achieved by code that is changed on average 2.5 times per month or 19.4 times per month. Based on these observations, and studies such as [20]-[22], which argue that the threshold for statistical significance should be lowered from 0.05 to 0.005, we reject both the hypothesis frequently-changed code has higher quality and the results obtained by our investigation which suggest the opposite.

**FIGURE 13: Quality vs Change-Frequency**



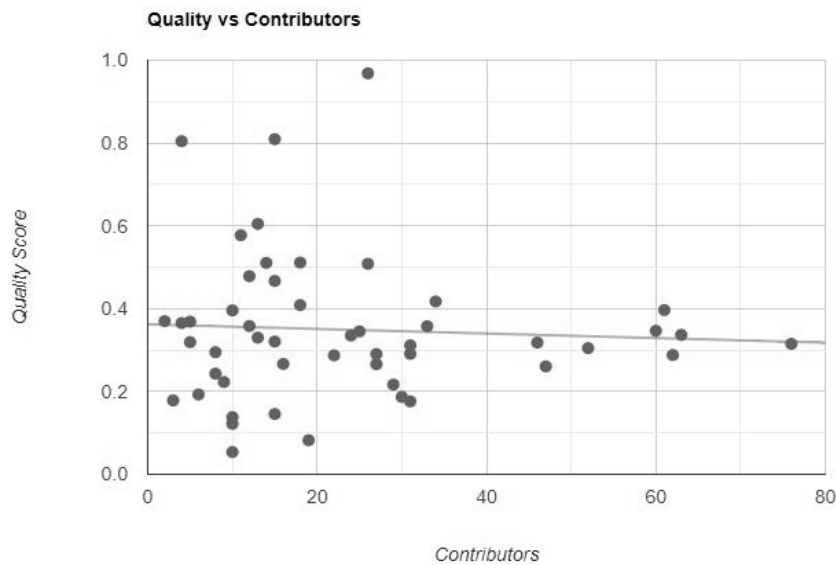
Correlation Coefficient: -0.2864  
(p-value ~ 0.04373)

#### 6.4 Code Quality vs Number of Contributors

H4 is a null hypothesis, based on previous related work, which asserts that there is no relationship between code quality and the number of contributors. This hypothesis is based on work by Norick et al. [9], who tested the effect of the number of contributors on surrogate measures of code quality. This project furthers their research by using a more comprehensive measure of code quality and extending their research from C++ projects to Java projects. Despite the different approach, the same null hypothesis was expected.

The number of contributors for each of the 50 projects across the 3-year period was counted, and the median quality score was calculated for each repository. The results were plotted on a scatter graph as shown in **FIGURE 14**. The graph shows no correlation between contributors and code quality, and statistical analysis supports this. The Spearman's Rank calculation produced a correlation coefficient of approximately -0.03, signifying no relationship between the two attributes. Therefore, the null hypothesis that code quality is not affected by the number of contributors can be accepted.

**FIGURE 14: Quality vs Contributors**



Correlation Coefficient: -0.0281  
(p-value ~ 0.84633)

## 6.5 Conclusion

The objective of this project was to investigate the code quality of open-source software and the factors that affect it. A weighted equation was formed to calculate a quality score using existing metrics, and quality scores were tested against possible influential factors. A purpose-built application was developed to facilitate the extraction and processing of the necessary data and information from the GitHub API. The application also incorporates additional functionality which allows the user to understand more about the quality of the code they analyse, including measures of technical debt and code smells. All of the acceptance criteria and use cases defined in the requirements specification have been satisfied and verified (see sections 2.0 and 5.2 respectively).

Fifty Java projects were selected randomly from a list of GitHub projects that matched pre-defined criteria, and their source code quality was measured using a custom equation based on metrics from the object-orientated CK Metrics suite. Data about each repository was extracted from the GitHub API and compared to code quality to investigate the influence of various factors on code quality. Four hypotheses were formulated regarding the impact of time, code size, change-frequency and the number of contributors on code quality, which this research aimed to either validate or disprove. Three out of four hypotheses were validated.

The research found that code quality is negatively affected by time and increasing code size (H1, H2), although quality generally decreased trivially due to these factors. Increasing code size was found to be correlate more strongly with the loss of code quality than time.

H3 investigated the influence of change-frequency on code quality and theorised that quality would be positively impacted by higher change-frequency. This hypothesis was found to be invalid, with the results of the investigation suggesting that quality was worsened by high change-frequency, albeit with debatable statistical significance.

The final null hypothesis built upon existing work which found that the number of contributors has no effect on code quality. This project extended this work by using recognised metrics as a more comprehensive measure of code quality and analysing a different programming language. However, the outcome of this research was consistent with the previous work, and the null hypothesis was accepted, that the number of contributors does not affect code quality.

The significance of this research relates directly to the software development process. This study has proven that increasing code size is more likely to have a detrimental effect on quality than time, change-frequency or number of contributors, which translates to adopting a methodology of writing more concise code to maximise code quality. The null hypothesis proven in this research that code quality is not affected by the number of contributors is also significant. Managers can be less cautious of trying to streamline the size of their development team to avoid adding complexity and poor-quality code to their product.

This project has also discovered several opportunities for future work. Further research could be conducted to discover the more specific reasons for the loss of code quality over time, and investigate the effect of change-frequency on code quality in greater detail, for example, by suggesting an optimal threshold for monthly change-frequency which maximises code quality. The weighted equation used to calculate code quality in this project could also be modified to assign greater importance to preferred aspects of code quality over others, and investigate the resultant quality scores against the same factors to discover whether the factors have different effects on individual aspects of code quality. For example, if the equation was modified to penalise complexity less, does code size have less of a negative influence on quality?

## 7.0 References

- [1] D. Spinellis, *Code Quality: The Open Source Perspective*. 2006.
- [2] G. Cristina Venera *et al*, "INFLUENCE FACTORS FOR THE CHOICE OF A SOFTWARE DEVELOPMENT METHODOLOGY," *Accounting and Management Information Systems*, vol. 10, (4), pp. 479, 2011. Available: <https://search.proquest.com/docview/928928425>.
- [3] Jan Bartoníček, "Programming Language Paradigms & The Main Principles of Object-Oriented Programming," *CRIS - Bulletin of the Centre for Research and Interdisciplinary Study*, vol. 2014, (1), pp. 93-99, 2014. Available: <http://www.degruyter.com/doi/10.2478/cris-2014-0006>. DOI: 10.2478/cris-2014-0006.
- [4] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Tse*, vol. 20, (6), pp. 476-493, 1994. Available: <https://ieeexplore.ieee.org/document/295895>. DOI: 10.1109/32.295895.
- [5] S. R. Chidamber, D. P. Darcy and C. F. Kemerer, "Managerial use of metrics for object-oriented software: an exploratory analysis," *Tse*, vol. 24, (8), pp. 629-639, 1998. Available: <https://ieeexplore.ieee.org/document/707698>. DOI: 10.1109/32.707698.
- [6] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *The Journal of Systems & Software*, vol. 23, (2), pp. 111-122, 1993. Available: <https://www.sciencedirect.com/science/article/pii/016412129390077B>. DOI: 10.1016/0164-1212(93)90077-B.
- [7] GitHub. (2019). *GitHub features: the right tools for the job*. Available: <https://github.com/features>.
- [8] G. JAY *et al*, "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship," *Journal of Software Engineering and Applications*, vol. 2, (3), pp. 137-143, 2009. DOI: 10.4236/jsea.2009.23020.
- [9] B. Norick *et al*, "Effects of the number of developers on code quality in open source software," in Sep 16, 2010, pp. 1.
- [10] B. Rogers, "These Are the Top Languages for Enterprise Application Development," Aug 21, 2018.
- [11] Octoverse, "Top languages over time," 2018.
- [12] Z. C. Ani *et al*, "Evaluating the Quality of UCP-Based Framework using CK Metrics," *International Journal of Advanced Computer Science and Applications*, vol. 9, (11), pp. 642-631, 2018. . DOI: 10.14569/IJACSA.2018.091188.
- [13] M. R. J. Qureshi and W. A. Qureshi, "Evaluation of the Design Metric to Reduce the Number of Defects in Software Development," *International Journal of Information*

*Technology and Computer Science*, vol. 4, (4), pp. 9-17, 2012. Available: <https://arxiv.org/abs/1204.4909>. DOI: 10.5815/ijitcs.2012.04.02.

[14] C. E. Cruz Ana, "Chidamber and Kemerer Suite of Metrics," unpublished, private communication, "May, 2008".

[15] R. Shatnawi, "A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems," *Tse*, vol. 36, (2), pp. 216-225, 2010. Available: <https://ieeexplore.ieee.org/document/5383377>. DOI: 10.1109/TSE.2010.9.

[16] K. A. M. Ferreira *et al*, "Identifying thresholds for object-oriented software metrics," *The Journal of Systems & Software*, vol. 85, (2), pp. 244-257, 2012. Available: <https://www.sciencedirect.com/science/article/pii/S0164121211001385>. DOI: 10.1016/j.jss.2011.05.044.

[17] Code Climate (2011). *Code Climate: Maintainability*. Available: <https://docs.codeclimate.com/docs/maintainability>.

[18] M. Aniche, "Code metrics for Java code by means of static analysis," Oct 5, 2015.

[19] G. JAY *et al*, "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship," *Journal of Software Engineering and Applications*, vol. 2, (3), pp. 137-143, 2009. DOI: 10.4236/jsea.2009.23020.

[20] J. P. A. Ioannidis, "The Proposal to Lower P Value Thresholds to .005," *Jama*, vol. 319, (14), pp. 1429-1430, 2018. Available: <http://dx.doi.org/10.1001/jama.2018.1536>. DOI: 10.1001/jama.2018.1536.

[21] D. J. Benjamin *et al*, "Redefine statistical significance," *Nature Human Behaviour*, vol. 2, (1), pp. 6-10, 2018. Available: <https://www.ncbi.nlm.nih.gov/pubmed/30980045>. DOI: 10.1038/s41562-017-0189-z.

[22] Cole Wayant, Jared Scott and Matt Vassar, "Evaluation of Lowering the P Value Threshold for Statistical Significance From .05 to .005 in Previously Published Randomized Clinical Trials in Major Medical Journals," *Jama*, vol. 320, (17), pp. 1813, 2018. Available: <https://www.ncbi.nlm.nih.gov/pubmed/30398593>. DOI: 10.1001/jama.2018.12288.



## 8.0 Appendix

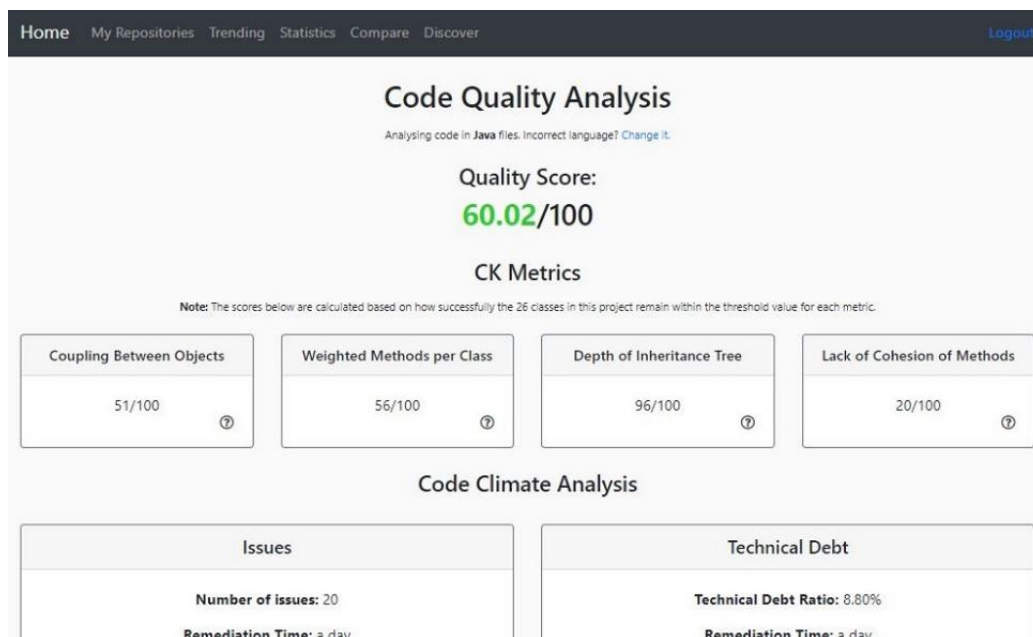
### Appendix 1. User Interface Design with regards to UX

The screenshot shows a web interface for archiving GitHub repositories. At the top is a navigation bar with links: Home, My Repositories, Trending, Statistics, Compare, Discover, and a Logout link. Below the navigation bar is a 'Welcome' section with a sub-header and a paragraph explaining the purpose of the page. A form is provided with a 'URL' label, a text input field containing 'GitHub Repository URL', and a 'Change form' button. Below the form are two date pickers labeled 'From:' and 'To:', both with placeholder text 'DD/MM/YYYY'. A green 'Archive' button is positioned below the date pickers. At the bottom, there is a section titled 'Repository Name' with four cards, each displaying a date (2019-04-01, 2019-03-01, 2019-02-01, 2019-02-01) and a laptop icon with code symbols.

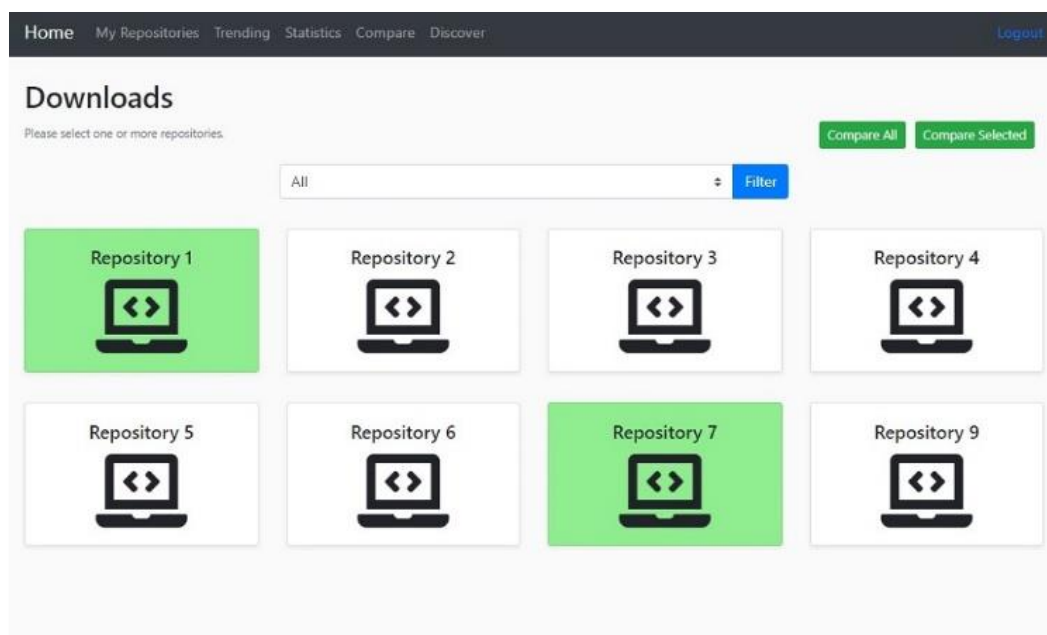
Simplistic pages with a consistent and professional design. Navigation bar present on every page.

This screenshot is similar to the previous one, but with a date picker widget open over the 'From:' date field. The date picker shows the month of April 2019, with days of the week (Su, Mo, Tu, We, Th, Fr, Sa) and dates (1-31). The date '19' is highlighted. The 'To:' date field is also visible, with placeholder text 'DD/MM/YYYY'. The 'Repository Name' section at the bottom is partially obscured by the date picker.

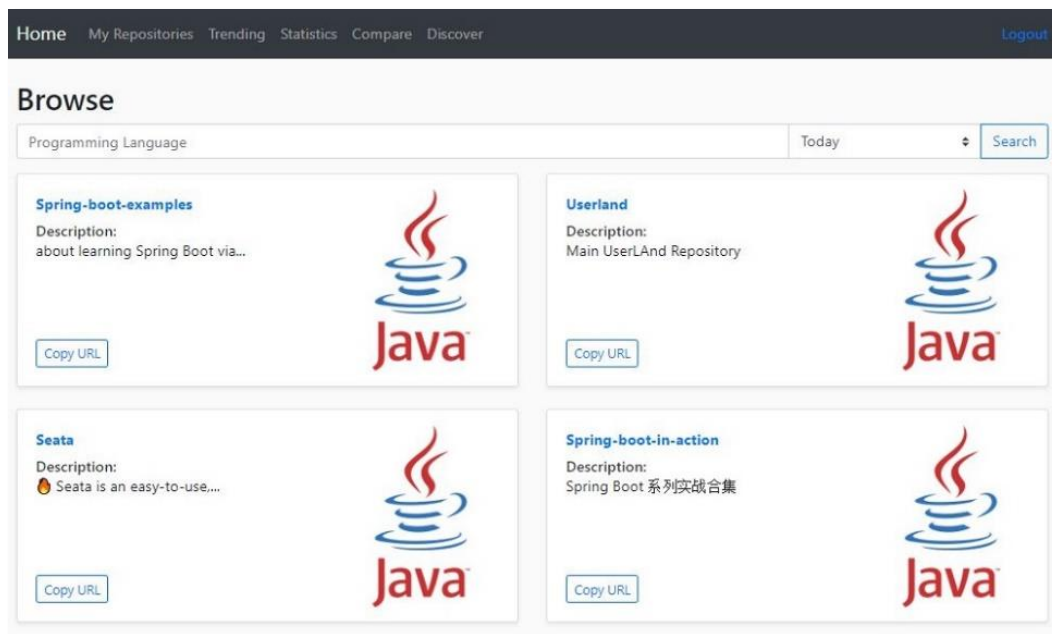
Date picker widget used to assist user input. Form validation included with descriptive error messages.



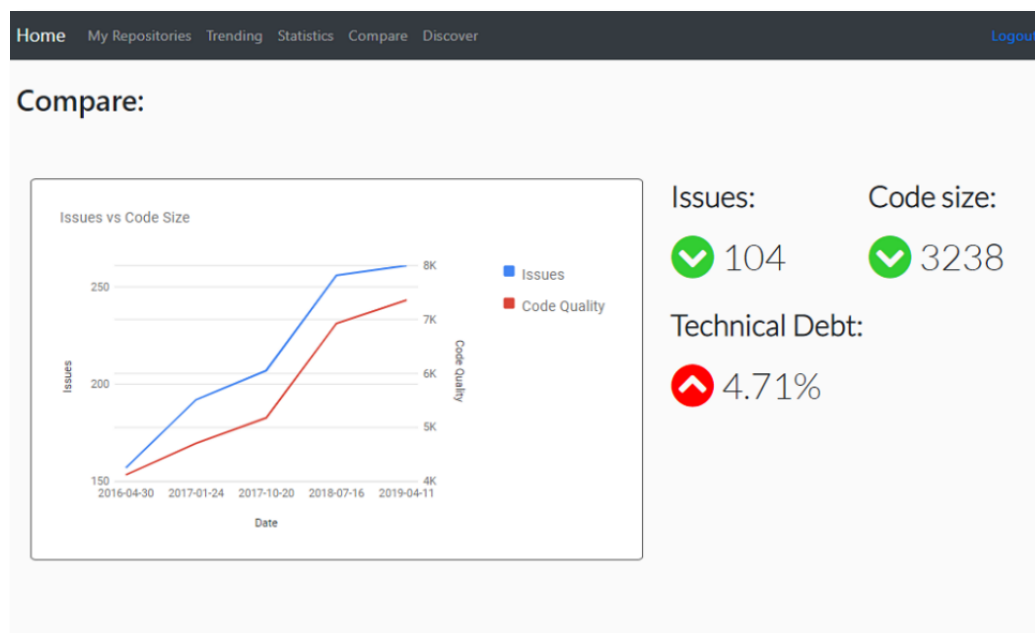
Key information highlighted to signify importance.



Use of AJAX to perform filtering/searching to minimise user wait time.



User-friendly shortcuts for copying repository URLs.



Use of colour scheme to indicate improvement or otherwise. Visualisations for intuitive interpretation of differences between repositories.

## Appendix 2. Wireframes

### Login

A wireframe for a login page. At the top, a browser window header shows 'Welcome Page' and a search bar with 'https://'. Below the header is a navigation bar with links: Home, My Repositories, Trending, Statistics, Compare, and Discover. The main content area features a large 'Welcome' text, a circular GitHub logo, and a 'Login' button. A footer bar is at the bottom.

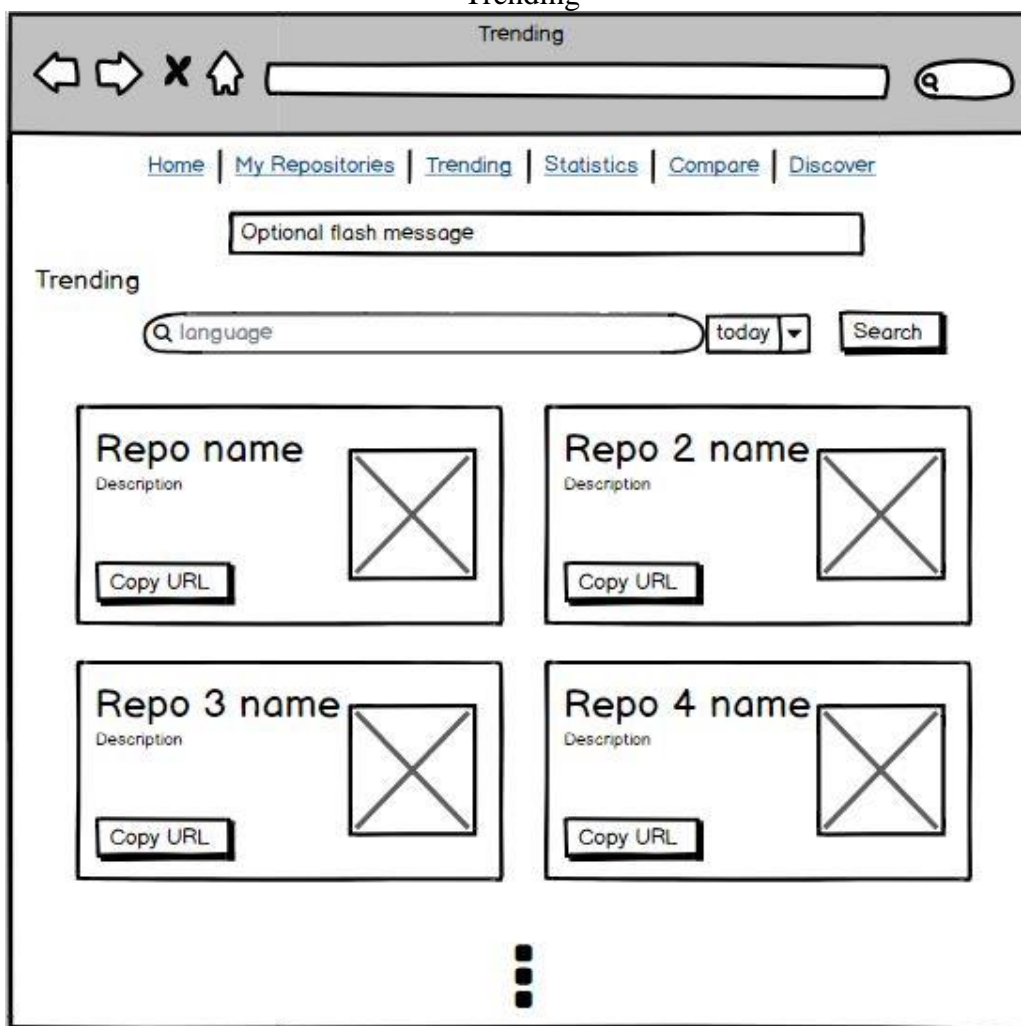
### Home Page

A wireframe for a home page. The browser window header shows 'Home Page' and a search bar. The navigation bar is identical to the login page. Below the navigation bar is an 'Optional flash message' box. The main content area starts with 'Welcome, <username>'. Below this is a 'Repository URL' input field with a 'Change form' button. A date range selector shows 'From' and 'To' with calendar icons and a plus sign between them. Below the date range is an 'Archive' button. The main content area then displays a list of 'Archived Repository' items. Each item has a title and three date buttons (2001-01-01, 2002-01-01, 2003-01-01). Below each set of date buttons is a link that says 'Compare all <repo-name> versions'. The list shows 'Archived Repository 1', 'Archived Repository 2', and 'Archived Repository n' with vertical ellipsis between the second and third items.

### My Repositories



### Trending



## Discover

Discover

[Home](#) | [My Repositories](#) | [Trending](#) | [Statistics](#) | [Compare](#) | [Discover](#)

Optional flash message

Discover

Filters:  
Filter 1 | Filter 2 | Filter 3  
[change filters](#)

Select All | Copy Selected

Select Repositories:

Repo name	Repo name	Repo name
Repo name	Repo name	Repo name
Repo name	Repo name	Repo name

## Change Discover Criteria

Change Filters

[Home](#) | [My Repositories](#) | [Trending](#) | [Statistics](#) | [Compare](#) | [Discover](#)

Optional flash message

Set Filters

Language

Min size

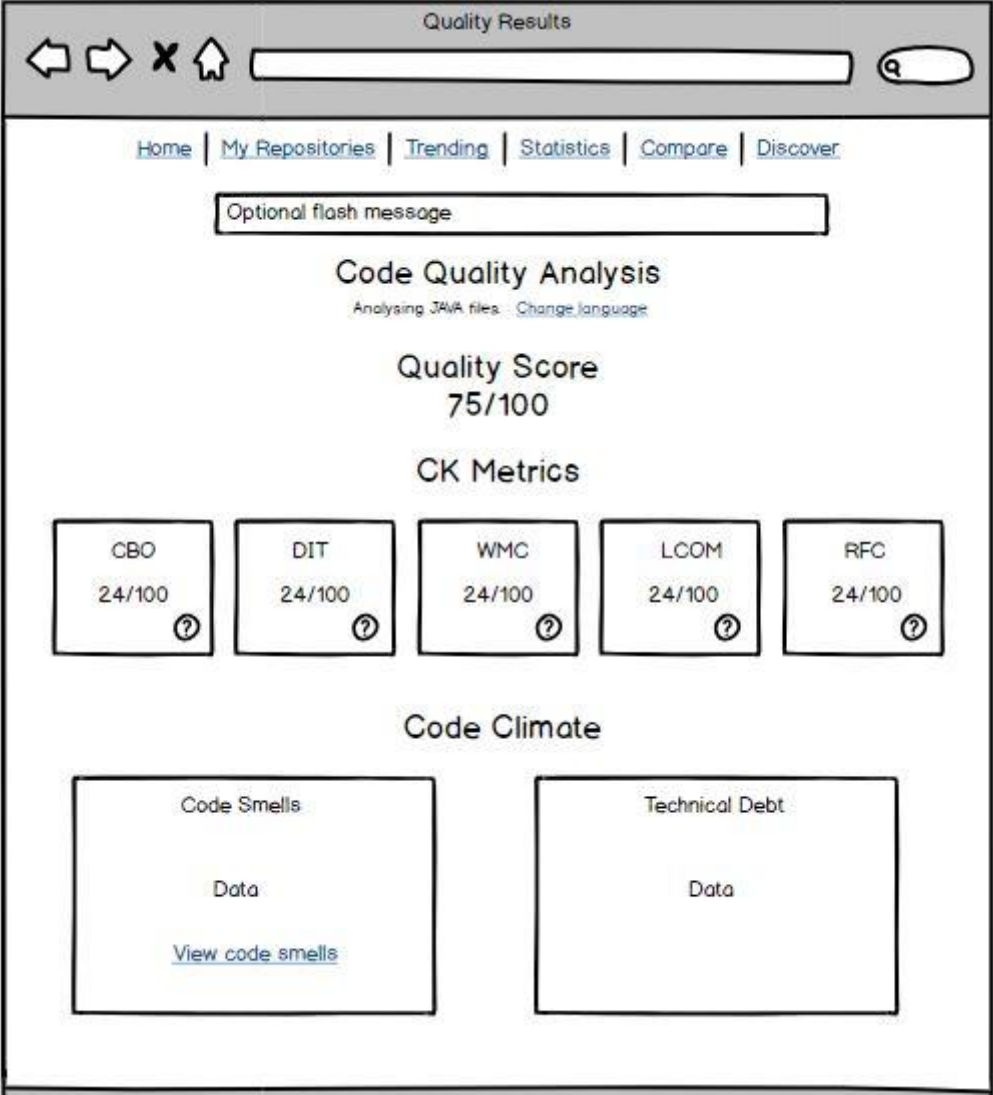
Max size

Created before

Updated after

Submit

## Quality Analysis (single repository snapshot)



Quality Results

Home | My Repositories | Trending | Statistics | Compare | Discover

Optional flash message

### Code Quality Analysis

Analysing JAVA files. [Change language](#)

#### Quality Score

75/100

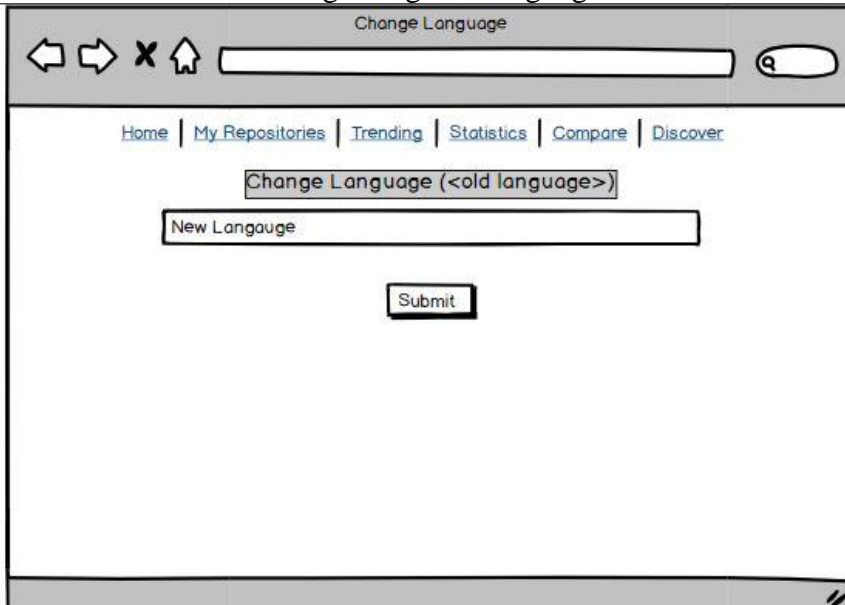
#### CK Metrics

CK Metric	Score
CBO	24/100
DIT	24/100
WMC	24/100
LCOM	24/100
RFC	24/100

#### Code Climate

Code Climate	Data
Code Smells	<a href="#">View code smells</a>
Technical Debt	

## Change Linguist Language



Change Language

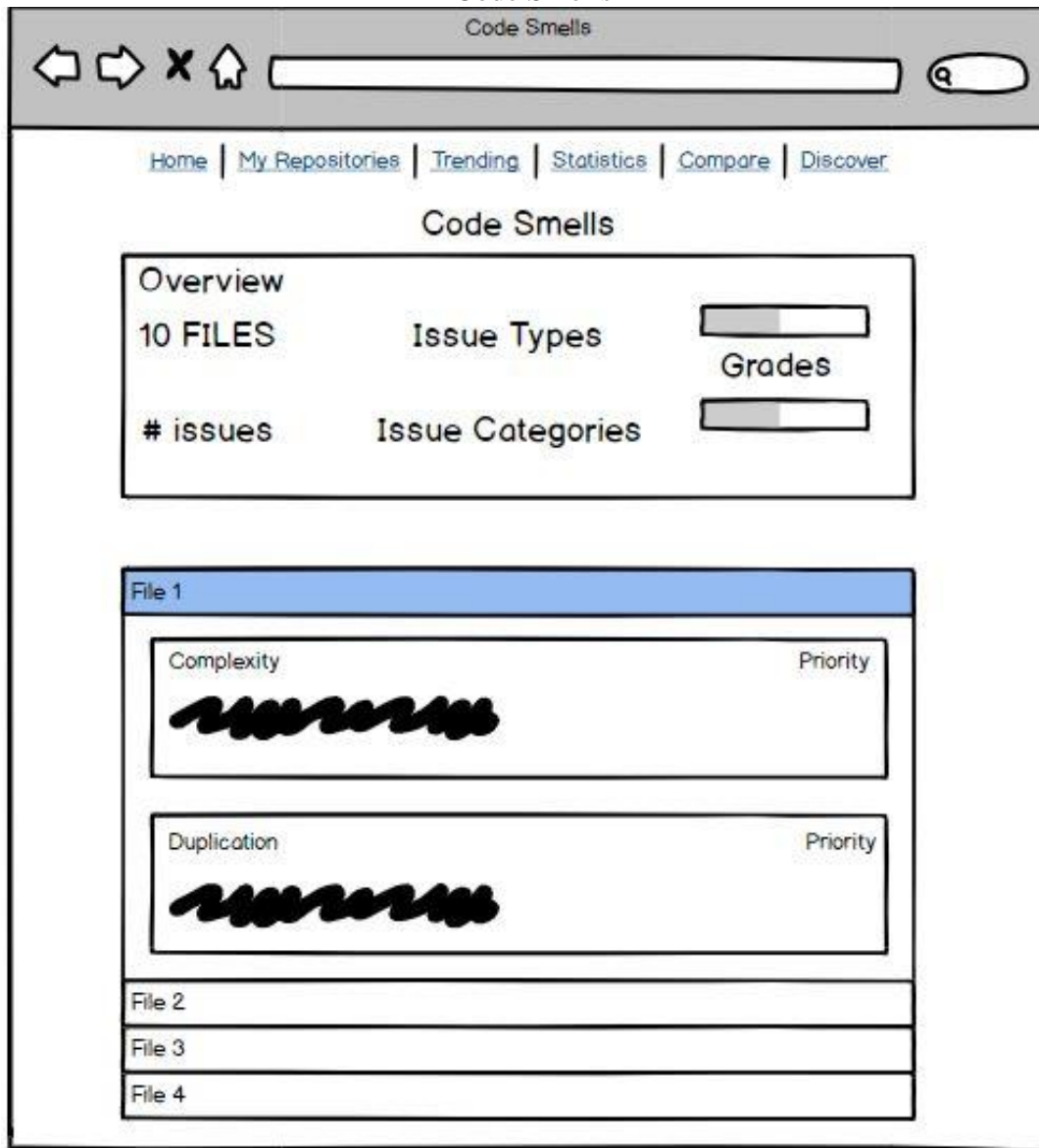
Home | My Repositories | Trending | Statistics | Compare | Discover

Change Language (<old language>)

New Language

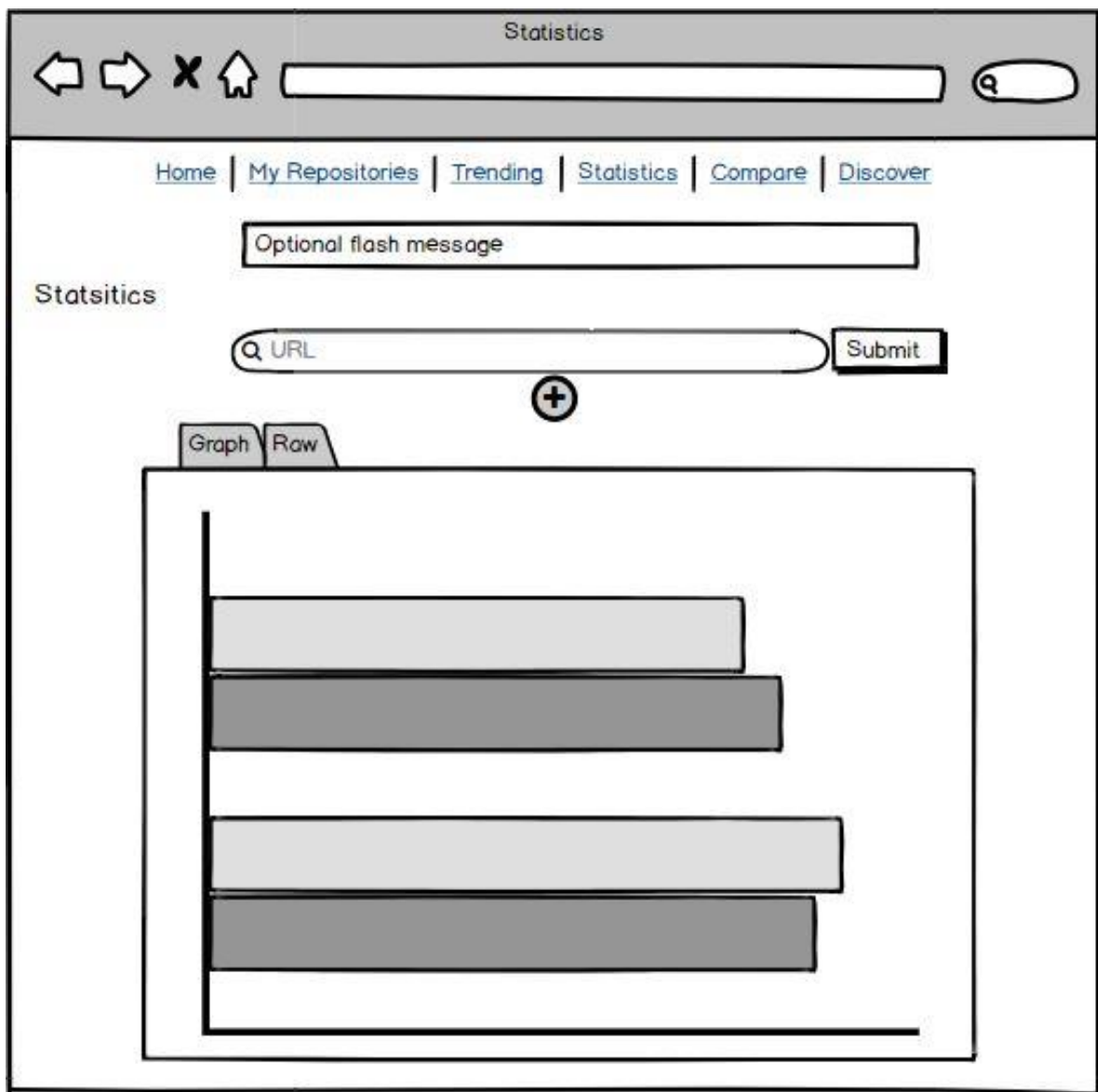
Submit

## Code Smells

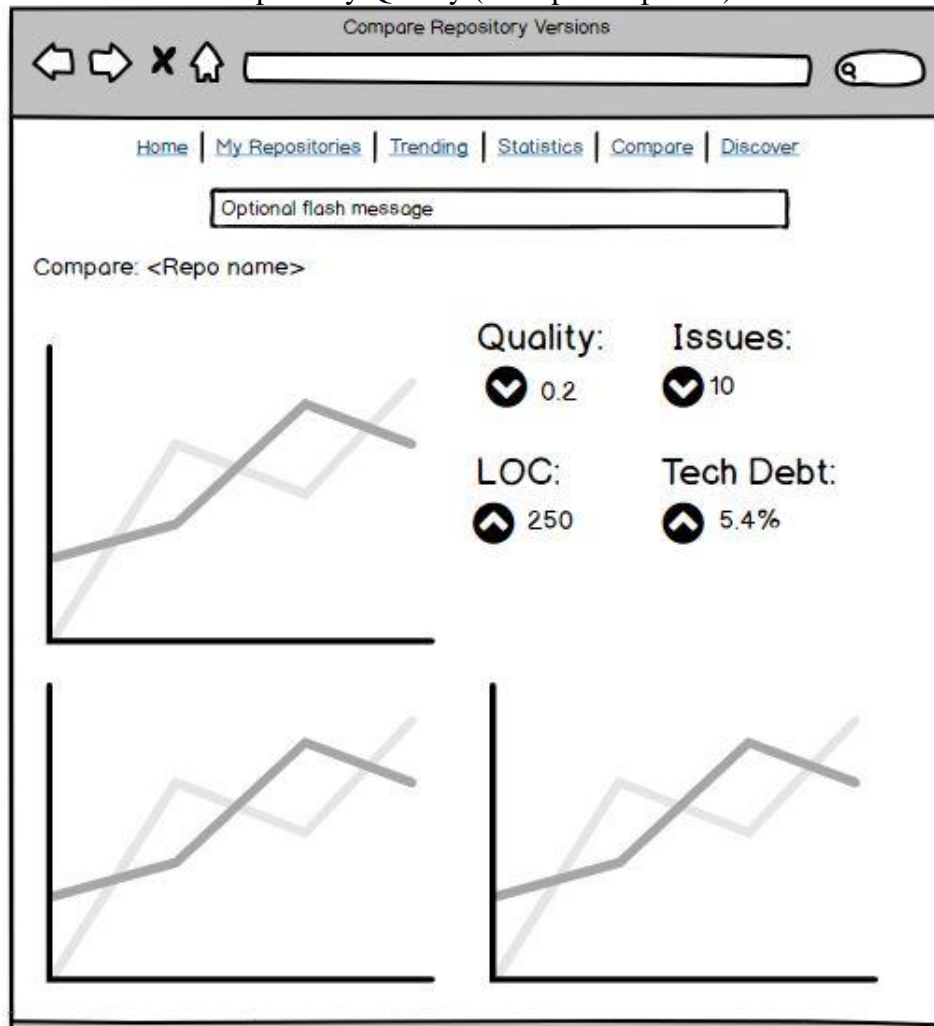




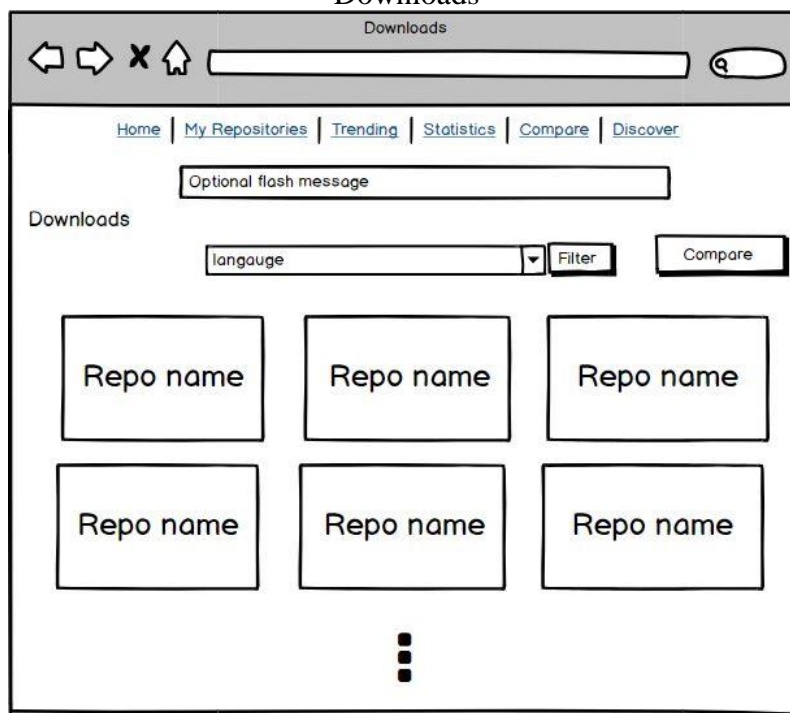
## Statistics



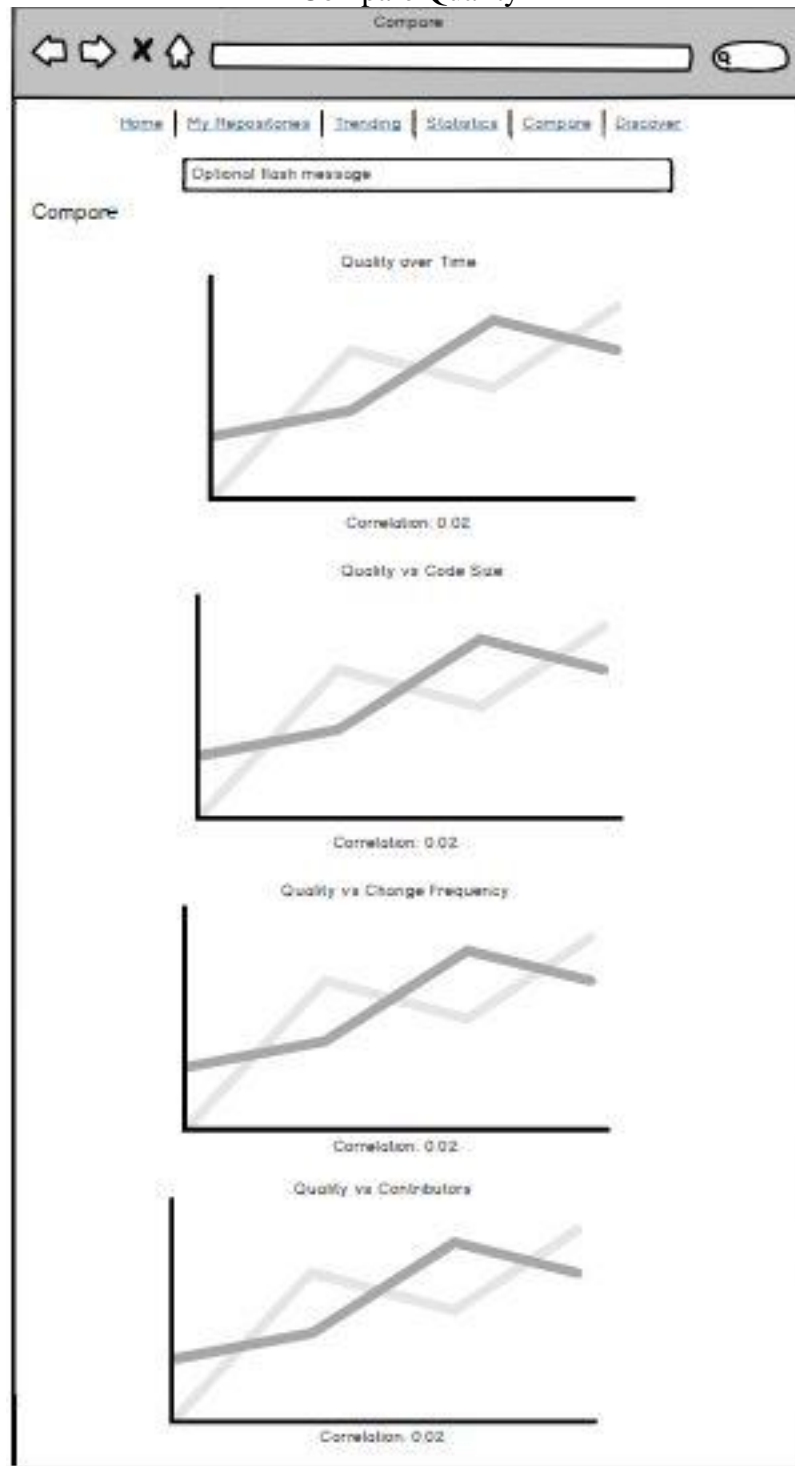
## Repository Quality (multiple snapshots)



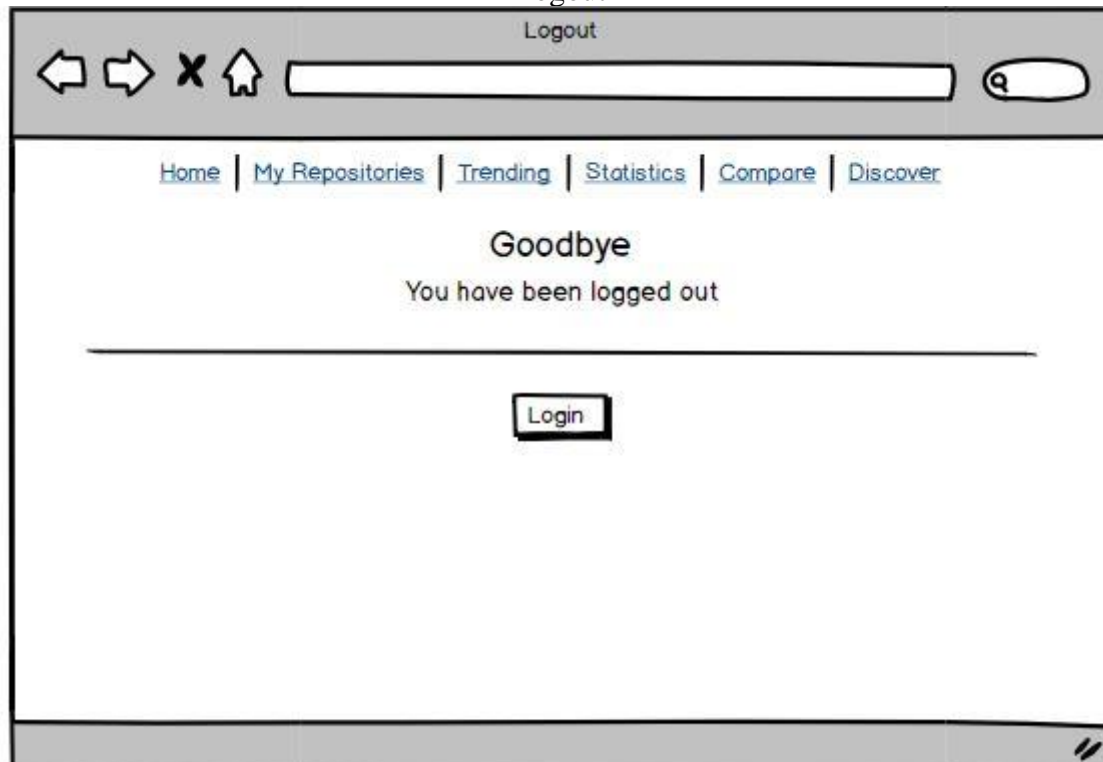
## Downloads



## Compare Quality



## Logout



### Appendix 3. Unit Test Coverage

Name	Cover
code_climate.py	100%
helpers.py	100%
jinja_templates.py	100%
routes.py	98%
tasks.py	100%
TOTAL	99%

### Appendix 4. Unit Test Results

```
(csproject)$ coverage run tests/test_jinja_templates.py
.....
-----
Ran 15 tests in 0.616s
OK
```

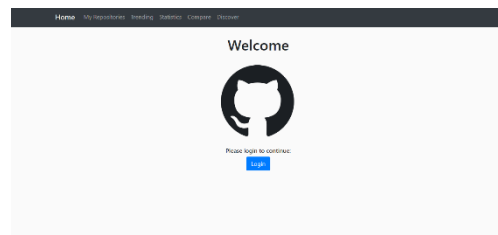
```
(csproject)$ coverage run tests/test_helpers.py
.....
-----
Ran 36 tests in 8.869s
OK
```

```
(csproject) $ coverage run tests/test_routes.py
.....
-----
Ran 28 tests in 8.036s
OK
```

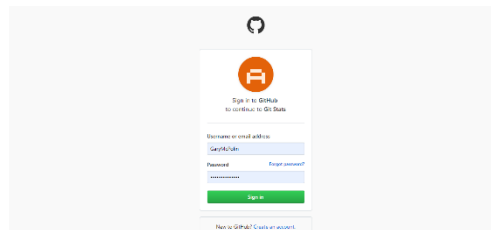
```
(csproject)$ coverage run tests/test_tasks.py
.....
-----
Ran 24 tests in 10.616s
OK
```

```
(csproject) $ coverage run tests/test_code_climate.py
.....
-----
Ran 24 tests in 11.978s
OK
```

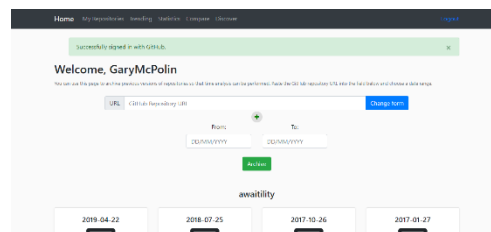
### Appendix 5. User Login Acceptance Test



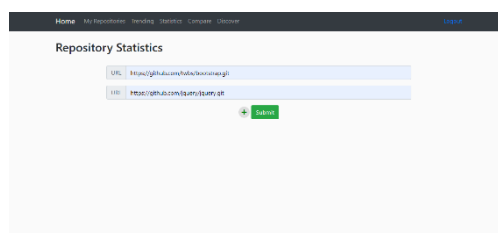
Redirect to GitHub OAuth login



Login successful with GitHub credentials



Appendix 6. Visualise GitHub statistics for different projects Acceptance Test



Submit the form with 2 repository URLs



View statistics as a graph

Repository Statistics

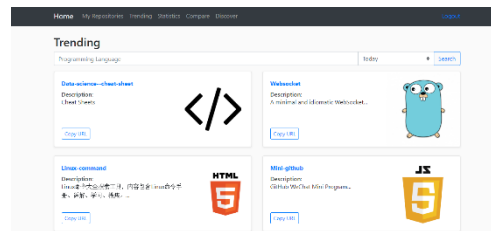
GitHub Repository Stats

Stats New

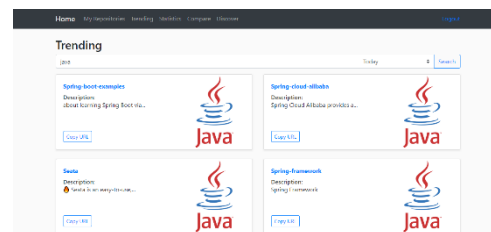
Repository		Branches	
Repositories	11117	Repositories	11117
Stars	4	Stars	48
Issues	2459	Issues	122
Wiki	1000	Wiki	1000
Open Issues	12	Open Issues	496
Open Pulls	11447	Open Pulls	11117
Size	1000	Size	11117

View statistics as raw values

## Appendix 7. View trending repositories (filter by language) Acceptance Test

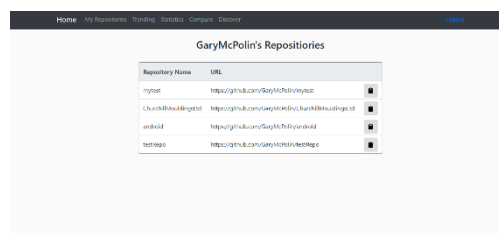


Navigate to trending page. Filter by Java

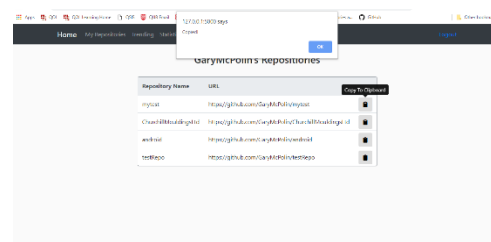


Java repositories displayed

## Appendix 8. Access quick-links to user repositories Acceptance Test



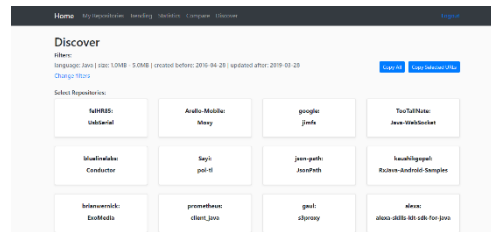
Navigate to my repositories page. Click clipboard icon to copy URL



URL copied

## Appendix 9. Find GitHub repositories for analysis Acceptance Test

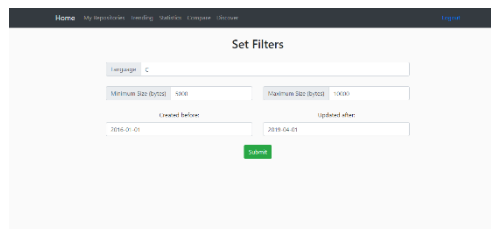
Navigate to discover page.



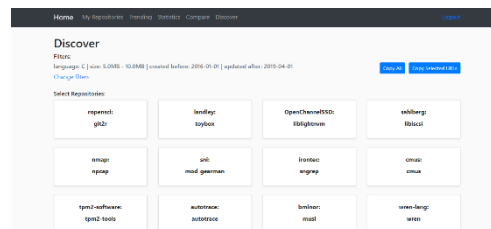
Observe repositories found using default criteria

## Appendix 10. Find GitHub repositories for analysis using custom search criteria Acceptance Test

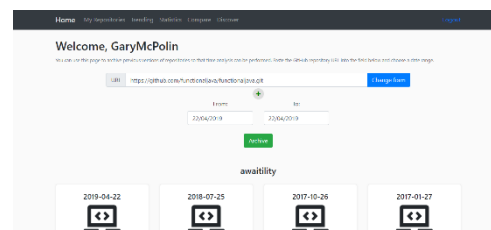
Navigate to discover page. Click change filters. Submit form with custom filters.



Observe new list of repositories which match search criteria. Ensure search criteria is visible at the top of the page.

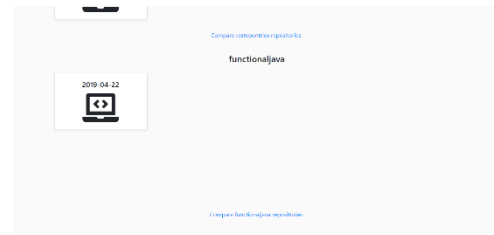
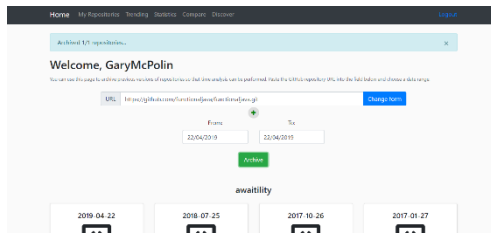


## Appendix 11. Analyse Source Code for a single version of one repository Acceptance Test



Submit form with same date for “from” and “to” fields. Observe success message after archive finishes.

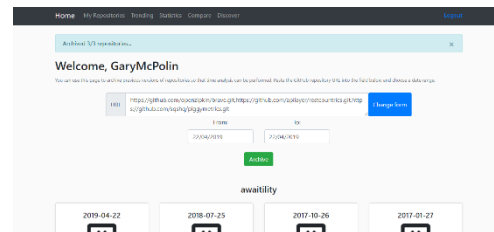
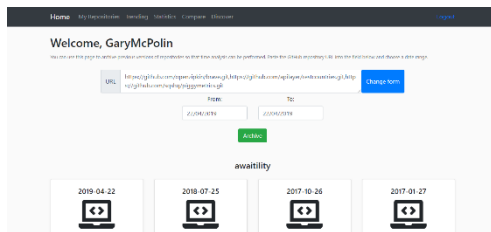




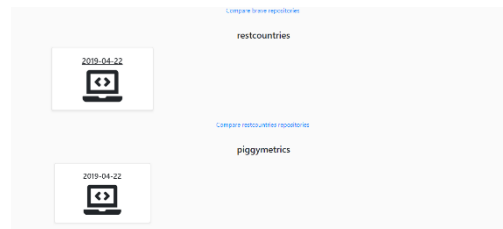
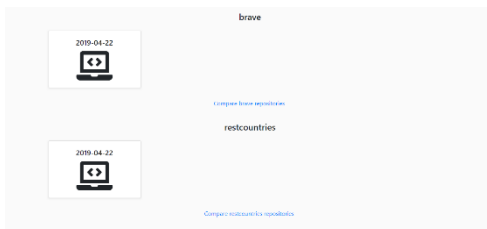
Single version archived.

## Appendix 12. Analyse Source Code for a single version of multiple repositories Acceptance Test

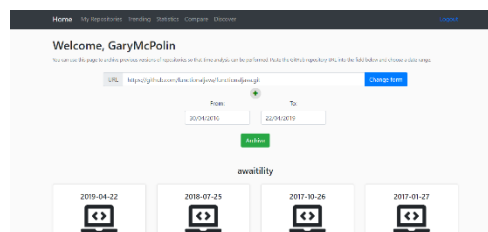
Submit form with 3 repository URLs same date for “from” and “to” fields. Observe success message after archive finishes.



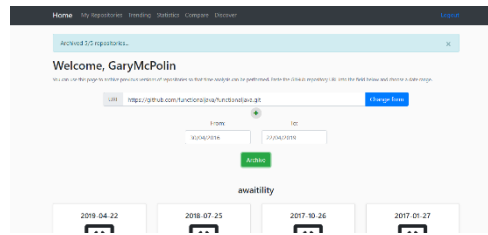
Single version of 3 repositories archived.



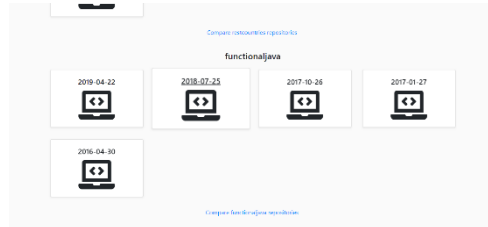
## Appendix 13. Analyse Source Code for multiple versions of a single repository Acceptance Test



Submit form with different dates for “from” and “to” fields. Observe success message after archive finishes.

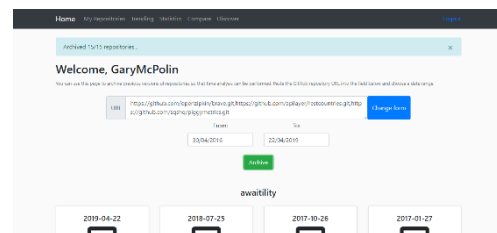
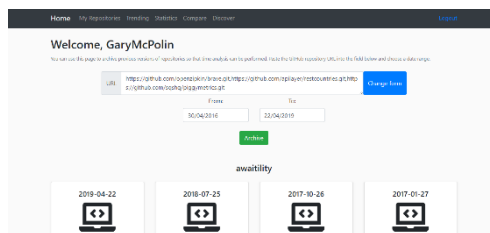


Multiple versions of single repository archived.

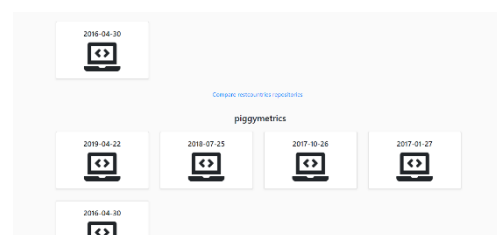
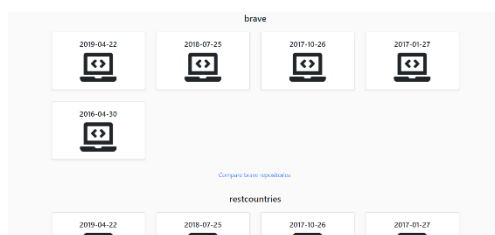


Appendix 14. Analyse Source Code for multiple versions of multiple repositories  
Acceptance Test

Submit form with different dates for “from” and “to” fields. Observe success message after archive finishes.

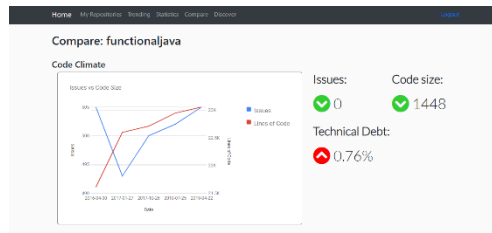


Multiple versions of 3 repositories archived.



Appendix 15. Visualise code quality and GitHub statistics for multiple versions of a single project  
Acceptance Test

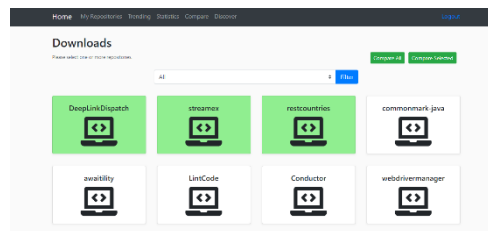
Navigate to compare repository versions page



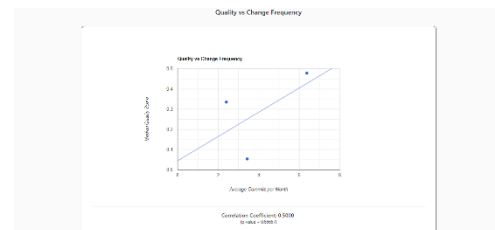
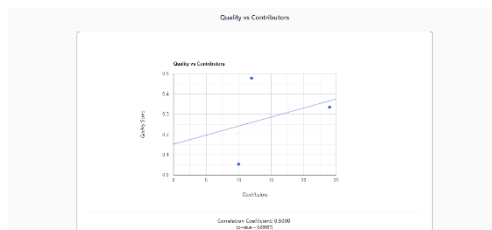
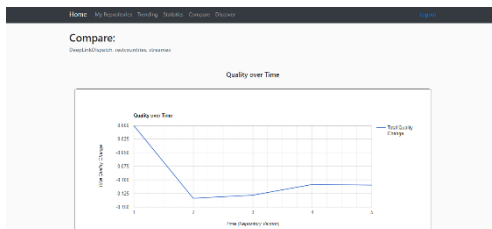
Observe graphs displayed and note 5 vertices on graph indicating all 5 versions of the repository are represented.

## Appendix 16. Visualise code quality and the factors affecting it for multiple versions of multiple projects Acceptance Test

Navigate to compare route and select 3 repositories to compare.

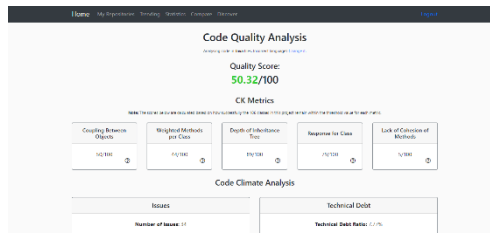


Click compare selected and ensure that all 3 repositories are represented on each of the 4 graphs

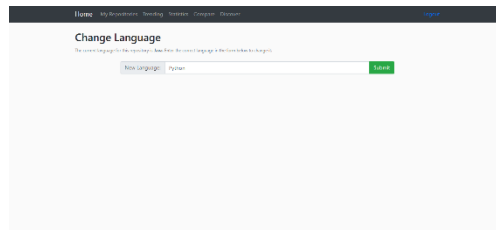


## Appendix 17. Change the default language of a repository Acceptance Test

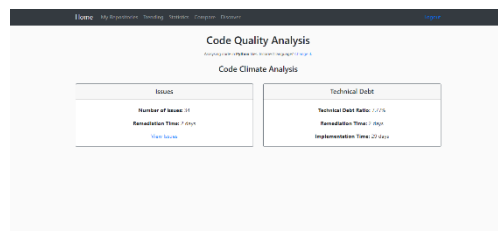
Navigate to the change language route. Observe the current language for the repository. Click change language.



Use the form to change the language to an alternative.

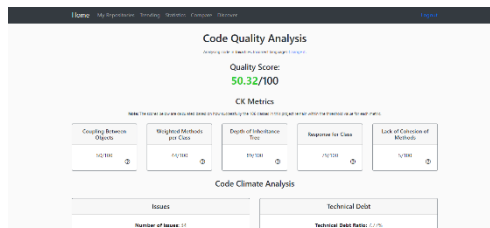


Navigate back to the change language route, ensuring that the language has been changed successfully.

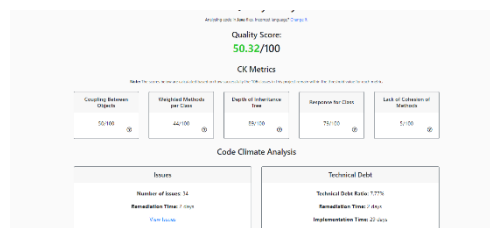


Appendix 18. View specific issues that are reducing code quality Acceptance Test

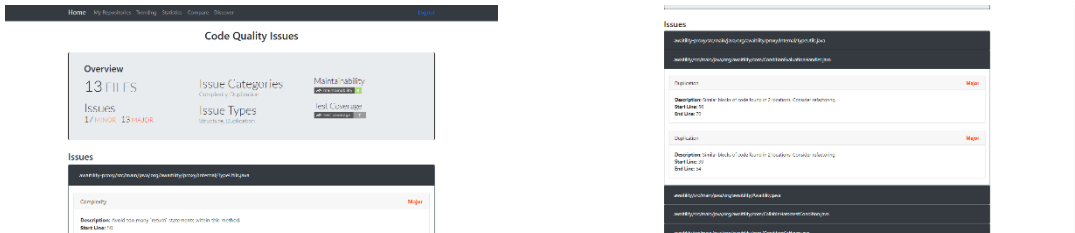
From the home page, click on a repository to analyse it.



Scroll to the bottom of the results page and click the view issues link.



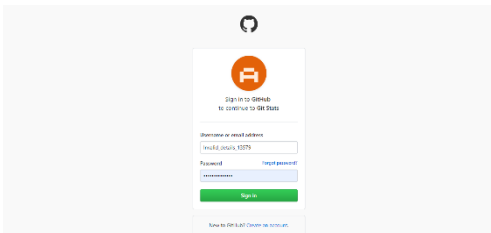
Ensure that issues for the project are listed and expand the accordion to view issues in different files.



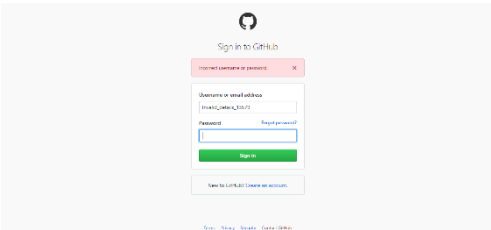
Appendix 19. Additional exemplar acceptance tests

Use Case	Requirement	Test Scenario	Pass Evidence
1	Unsuccessful login with invalid credentials	Navigate to the application home page as an unauthenticated user. Attempt to sign into the application using invalid GitHub credentials.	Appendix 20
2	Visualise GitHub statistics for a single project	Submit the form on the statistics page with one GitHub repository URL. View the graph and raw data.	Appendix 21
2	Visualise GitHub statistics with invalid URL	Submit the form on the statistics page with an invalid GitHub repository URL. Observe error message	Appendix 22

Appendix 20. Unsuccessful Login Acceptance Test



Invalid credentials → Login unsuccessful




Appendix 21. Visualise GitHub statistics for a single project Acceptance Test

[Home](#) [My Repository](#) [Searching](#) [Statistics](#) [Language](#) [Workflow](#) [Logout](#)

## Repository Statistics

URL

Submit



Submit the form with 1 URL



View statistics as a graph

Repository Statistics

1/1 2/2 in Repository (0)

Filter

Filter

Name	Size	Date
aws-iam	100MB	
aws-iam-iam	100MB	
aws-iam-iam-iam	100MB	
aws-iam-iam-iam-iam	100MB	
aws-iam-iam-iam-iam-iam	100MB	
aws-iam-iam-iam-iam-iam-iam	100MB	
aws-iam-iam-iam-iam-iam-iam-iam	100MB	
aws-iam-iam-iam-iam-iam-iam-iam-iam	100MB	
aws-iam-iam-iam-iam-iam-iam-iam-iam-iam	100MB	
aws-iam-iam-iam-iam-iam-iam-iam-iam-iam-iam	100MB	

View raw statistics

## Appendix 22. Visualise GitHub statistics invalid URL Acceptance Test

Submit form with invalid URL

[Home](#) [VS IDE/tutorial](#) [Installing](#) [Statistics](#) [Compare](#) [Overview](#) [Log out](#)

Repository Statistics

100%

GitHub repository URL

submit

Error message displayed

## Appendix 23. User Guide

1. Login
  - Click the login link on the application landing page
  - Log in using the GitHub portal by supplying your GitHub credentials
2. View your repositories and copy their URLs for analysis
  - Click on the “My Repositories” link on the navbar
  - Click the clipboard icon beside the name of the repository that you want to analyse
3. View trending (popular) repositories and copy their URLs for analysis
  - Click on the “Trending” link on the navbar
  - Click the “Copy URL” button beside the name of the repository that you want to analyse
  - You can filter the results by programming language or by daily, weekly or monthly trending repositories using the form provided at the top of the page.
4. Find repositories that match specific criteria and copy their URLs for analysis
  - Click on the “Discover” link on the navbar
  - Observe the default search criteria at the top of the page. If you wish, you can change these by clicking on the “Change filters” link.
  - Select one or more repositories and click the “Copy selected” button at the top of the page to copy the URLs. Alternatively, you can copy all URLs by clicking the “Copy All” button at the top of the page.
5. View statistics from the GitHub API about one or more repositories
  - Click on the “Statistics” link on the navbar
  - Fill out the form using one or more GitHub URLs and click submit
  - View the results as a graph or click the “Raw” tab to view the raw data
6. Analyse quality one or more repositories
  - Navigate to the application home page
  - Paste the repository URL(s) into the form provided and choose the dates over which to analyse the code. You can choose the same date to analyse a single version of the code. If you have many URLs, you can click the “change form” button and paste the URLs as a comma-separated string for convenience.
  - Wait for the archive job to complete – you will see a success message if it is successful.
  - Scroll to the version(s) of the newly-archived repository on the home page. Click either a single version to see information about its quality analysis, including score;

specific code smells and technical debt. Alternatively, if you downloaded multiple versions of the repository, you can click the “Compare <my repo> repositories” link to view the quality of the repository over each of the versions.

7. View how various factors influence code quality

- Click on the “Compare” link on the navbar
- Click on the repositories you are interested in and click “Compare Selected” or click “Compare All” to compare all repositories.
- Observe the graphs produced which show the effect of time, code size, change-frequency and number of contributors on the quality of the repositories chosen.

8. Logout:

- Click the logout button on the right side of the nav bar from anywhere within the application