Implementing A Bug Localization Service In An Educational Code Analysis System

Vo Ngoc Duy Nguyen[1]

Vietnam National University of Ho Chi Minh City, International University, School of Computer Science and Engineering vondnguyen@gmail.com

Abstract. Providing evaluation for performance is a crucial part of developing programs as well as in education. While teachers commonly use grading systems or test suites to evaluate students' performances as well as providing feedback, the results that test suites provided can yield vague results in some cases. As students can make mistakes in the development process, it can be difficult to track down and identify the problems occurring in their applications without any clear insight into the location of the errors and a proper way to handle them. This thesis introduces an approach on using a bug localization tool named FLACOCO, which utilizes a commonly used bug localization method called Spectrum-based Fault Localization (SBFL). SBFL uses code coverage to predict faulty lines of code and assign a suspicious score on each, thereby improving the feedback provided to students and giving an enhanced learning experience. The idea is to integrate this service into a previously implemented system that generate questions about learner's code , previously developed in a prior thesis. This integration aims to increase the precision and provide a form of feedback by pinpointing error locations within student submissions by using test cases provided by the teacher. However, SBFL tools in my knowledge, have not been widely adopted due to their low maturity. Through this thesis, I aim to offer insights into the practical application of this tool as well as demonstrating its limitation by results and experiment.

Keywords: Bug Localization · education · pedagogy · test cases · software engineering.

1 Introduction

1.1 Motivation

Providing a better understanding of programming and coding processes has always been a fundamental goal in computer science education. Despite of the fact that educators rely on a range of fundamental tools traditionally, from manual code reviews to automated grading systems to improve learning experiences; these methods often fall short in providing detail feedback which helps students to refine their learning curve, specifically in the debugging phase.

Recently, automated grading systems have become increasingly relevant in educational facilities. Computer assisted assessment systems are becoming increasingly common in higher education as aiming for accurate feedback on assignments as well as providing a thorough understanding of the programming process has been the identified as the key factor of successful learning among students. [14] The use of digital technologies presents educators with an opportunity to develop more interesting learning curve from many resources [12], not only from the internet but also from student's performance.

Educational assignments in computer science often adopt a one-size-fits-all approach, as a method of conveying foundational information by using larger scope exercises such as a big project. Tackling these assignments has been demonstrated to cultivate students with a sense of independent research and creative thinking ability [12]. However, this may not cater to various learning paces and styles across students. The fact that some educational institutions have already researched and implemented various form of an automated assessment system has shown that it is very time consuming and impractical in some cases to present extensive feedback on large amount of projects if done traditionally [14].

Automated grading systems offer a range of functionalities. They can check the functional correctness of assignment submissions as well as providing feedback back to students in real-time [8]. Moreover, the capability for detecting plagiarism and collecting data to assess students is also in scope. [14] Using these systems, students and educators can benefit from the data processed from submissions, allowing students to revise and learn from their mistakes. However, it can sometimes provide unclear details about the mistakes that students often encounter in the feedback. In the current settings, these systems rely on running the submission against a series of test cases whenever student upload their programming project submission, and the failing test cases are returned to students as a form of feedback [8]. Since failing test cases do not usually offer any guidance specifically, the location of the bugs and along with a proper way to handle them can be difficult for students to find.

1.2 Problem Statement

To aid the process of debugging by providing the student with an idea where the bug can be located, this thesis propose the integration of a bug localization service using a Spectrum-Based Fault Localization (SPFL) tool on a previously implemented analysis system that analyzes the source code of submitted assignment from students and generate questions about learner's code [19]. SPFL is a well-researched bug localization technique that utilizes the use of code coverage to identify the faulty line of code and predict the suspiciousness rating for that line.

Initially focusing on smaller, discrete programming exercises. The research will assess the feasibility of SBFL tools in assisting student understanding by providing information such as the tests found and ran, the location of the bug as well as the suspiciousness score in the debugging process. Upon exploring the

SBFL tool, this thesis will demonstrate the usage of the tool, the potential it brings and its limitations in the current settings through some real projects.

SBFL tools are still maturing in their development and applications. However, they presents a unique opportunity to contribute to the growth of educational technologies. By situating this research at the intersection between bug localization and automated learning, the thesis aims to presents new methodologies that could provide support into teaching programming, making the learning process both more effective and informative.

1.3 Scope and Objectives

This thesis focuses on integrating a bug localization service onto a previously implemented system that generates questions of learner's code called the called Learner Code Analyzer system (LCA system) [19]. To successfully implement the service, two requirements need to be met: Restoring the LCA system to a fully functional state and gaining an understanding of the inner workings of a bug localization tool for the implementation.

The LCA system comprises of 3 main components: The front-end user interface, the back-end API server to handle requests from the web page and a question analyzer engine. In order to successfully reconstruct the LCA system, research into web applications and its framework that the system is currently utilized needs to be done. Understanding the GraphQL API with having familiarity with Prisma, an Object Relational Mapping (ORM) that stores data in a PostgreSQL database, will serve as the back-end research for the web server. For the front-end, the NextJS framework is used. The question analyzer engine handles questions submitted by students and analyzed it by parsing the project into Abstract Syntax Trees (ASTs) and receiving analyzer modules for question generation [19].

Furthermore, practice and research into various type of bug localization applications and some software repairs applications must be done, as some of the software repairs application also utilizes a form of bug localization such as Astor [13]. Implementing a SBFL tool within the LCA system, developing a test environment for testing for the implemented bug localization service, conducting an experiment and evaluating the effectiveness of SBFL tool from a simple algorithmic exercises to a more complex, real-life project as well as assessing the limitations of the tool are all the scope of this thesis.

In this thesis, the exploration of the integration of a bug localization service into an educational tool, with the objective of providing an insight into software debugging and improving learning curves. The contributions for this thesis include:

- Integrating an SBFL tool into the LCA system
- Conducting an experiment to assess bug localization capabilities in projects

1.4 Structure of thesis

Based on my findings, the structure of my thesis includes six chapters

- **Chapter 1. Introduction:** The introduction of my thesis is in the current section. The motivation along with the problem statement and the objective of my thesis is present in this chapter
- **Chapter 2. Background and related work:** This chapter focuses on the related work that contributed to this thesis.
- **Chapter 3. Methodology:** The methodology behind the components for this thesis is explained in this chapter, including the plan for reconstructing the LCA system, and the process of searching and experimenting with a bug localization tool.
- **Chapter 4. Implementation and results:** This chapter summarize the design and the actual implementation of the LCA system incorporating the bug localization service mentioned above.
- **Chapter 5. Discussion and evaluation:** In this chapter, the result for the experimentation is left for evaluation. The potential and the limitations of the localization tool will be discussed, as well as the reason for the selection of this tool for future growth.
- **Chapter 6. Conclusion and Future work:** This chapter will be about the result of the experimentation after the speculation from the previous chapter and then speaking about the foundational growth as a whole from this work for upcoming research endeavors.

2 Background And Related Work

2.1 Learner Code Analyzer System

The Learner Code Analyzer (LCA) system was developed as an intermediary platform between students and teachers to engage in meaningful discussions. These discussion are based on questions automatically generated from the programming assignment submission of students, which offers a dynamic educational tools that allows the enhancement for learning experiences as well as the flexibility of the learning curve for each students. [19]

The LCA system was designed with the thought of having the ability to be adaptable to changing the assessment requirements in mind [19]. Educators can develop their own customized analyzer, and they can contribute various analyzer into the system using a front-end user interface to contribute to its growth. As previously mentioned, the LCA system comprises of three main components:

- A front-end user interface for users and educators to interact with the assignment submissions and lexical analyzer modules.
- A back-end API server to handle requests from the front-end.
- An analyzer engine that runs the question generation by taking in the submitted solutions from the server and the analyzer modules created.

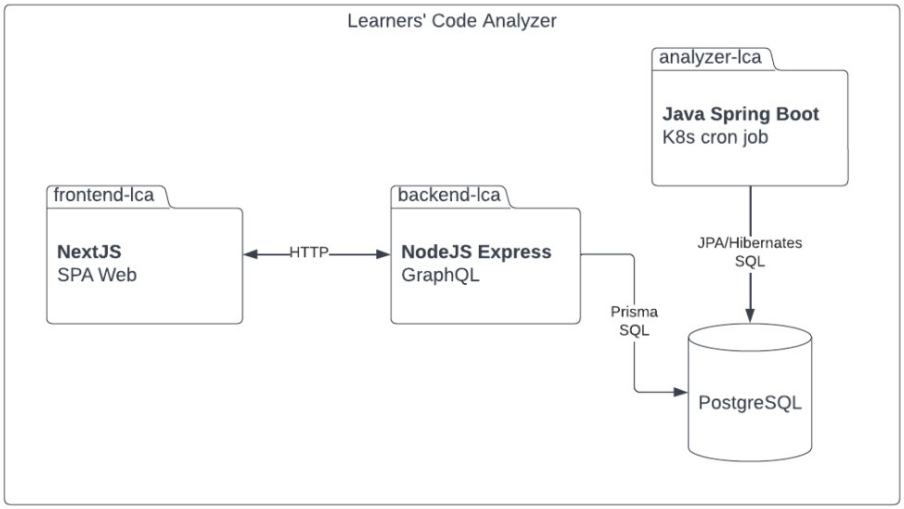


Fig. 2.1. The LCA System Architecture [19]

Analyzer engine Utilizing Spoon, which is a Java library that enables developers to work with and transform Java source code at a surface level through analysis. The process around the source code analysis stages using Spoon can be summarized as follows: Spoon works with the Java source code as an input, and then parse the code as a low-level abstract syntax tree. Then it changes the tree into a compile-time (CT) model which is more intuitive and easier to manipulate. The model is then processed by a user-defined "program processor" or "template" and then returned to the source. [17]

The mentioned user defined "template" above in the case of the LCA system is the lexical analyzer modules, which can be shown as the "custom analyzer storage" shown in **Figure 2.1**. The intended use of the analyzer storage depicted involves educators developing various types of lexical analyzers – like those for global variable or nested 'if' statement, classes, and more, all of which are supported in Java. The purpose for the LCA system is to utilize these pre-defined templates as a base for generating questions with the help of the Spoon Abstract Syntax Trees (AST) nodes, as illustrated in **Figure 2.3** and **Figure 2.4**.

Focusing on the pre-defined template inner workings and to explore the idea of how it operate. **Figure 2.4** describes in detail the process of how general custom analyzer module works. By traversing AST nodes to identify specific requirements based on pre-defined criterion using Spoon capabilities, the analyzer might come across simple or complex criteria such as the occurrence of static recursion functions that is used multiple times [19].

In details, a function containing nested loops and variables can be found by the analyzer; in the case of nested loops, the analyzer will filter out any loops that does not contain any other loops within. Or any variable declared

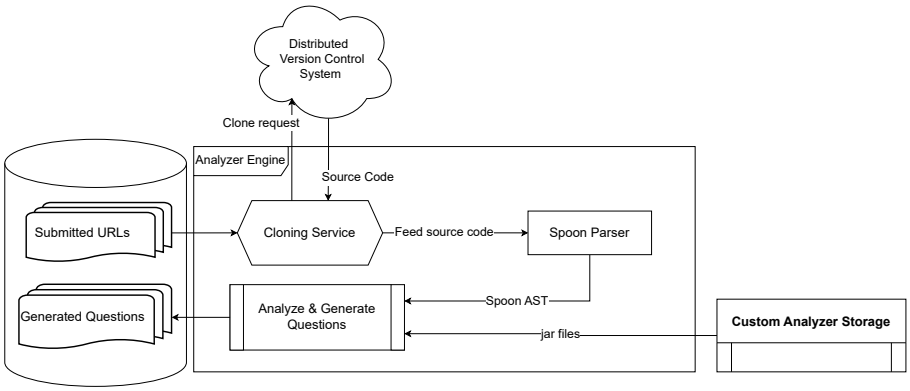


Fig. 2.2. A process of analyzing questions of the analyzer engine [19]

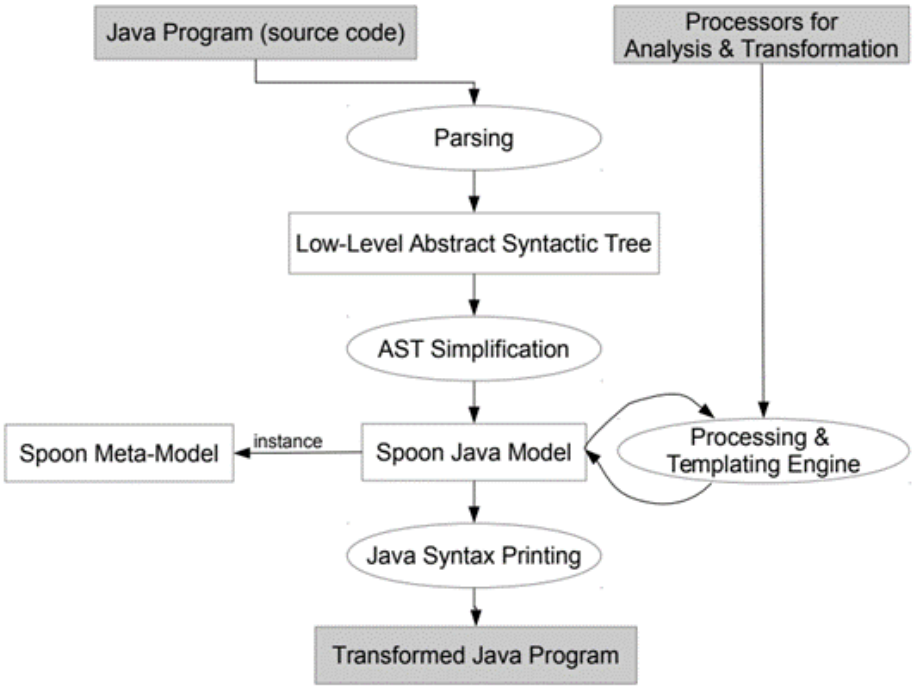


Fig. 2.3. An overview of the source code parsing process using SPOON [17]

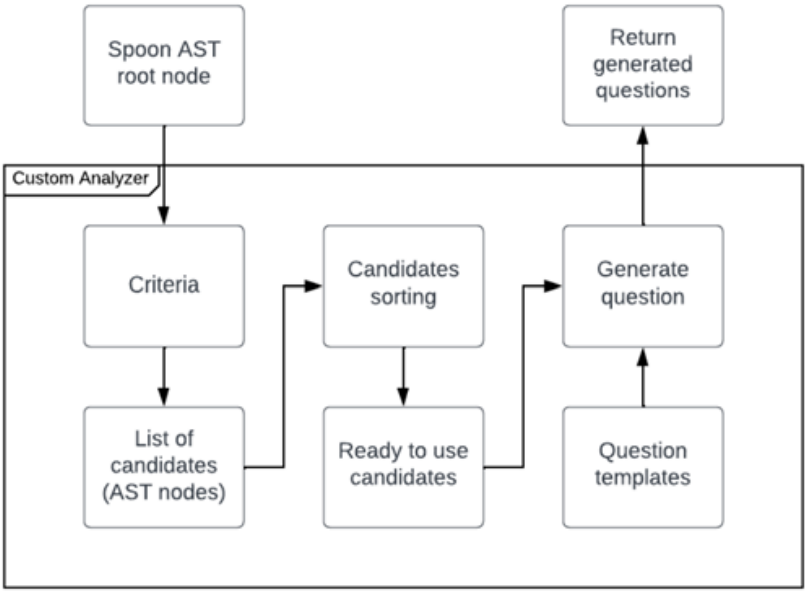


Fig. 2.4. An overview of a pre-defined template for generating questions [19]

outside of an internal function, or in the case of Java – any variable constants containing the 'static' or 'static final' keyword [25] will be selected as candidates. Once the candidates has been identified, they are sorted to determine their suitability for generating questions. The sorting process take account of many factors such as the cyclomatic complexity of the code elements, which assesses the computational flow complexity based on structures like loops and conditional statements. [19] After the suitable candidates have been selected, they are used with question templates to generate questions by matching the sorted criteria of the AST nodes to the appropriate question templates. By using Spring Data JPA [7]repository implemented in the analyzer engine, the questions are now saved in the PostgreSQL server.

Backend-LCA As shown in **Figure 2.1**, the system consists of a back-end system to receive and handle requests from the front-end. To distinguish the method of API handling from the analyzer, the back-end server does not use Spring Boot, particularly Hibernate SQL to handle requests from the front-end to interact with the relational database. Instead, it utilizes Prisma, an ORM which enables the use of a GraphQL API [6]. This GraphQL API plays the role of an intermediary between the user interface and the service, enabling the requests and response to be sent. [19]

Frontend-LCA The front-end employs NextJS – an open source web development framework that was built with NodeJS as a base. Along with Material UI, a React component library built by Google to help developers with fast features delivery. GraphQL queries and mutation are built to aid with the communication between the server side and the client side.

2.2 Bug Localization and the use of Spectrum-Based Fault Localization in Software Development

Bug Localization is the process of tracking and reporting back the specific location of the problematic code where defects or bugs might exist. Taking in a number of failing test cases and passing test cases, bug localization techniques can help developers to locate a specific area of the program to determine whether the bugs occurred in that region [18]. The technique is crucial in software development and debugging due to the reduced effort in locating bugs and addressing underlying issues in complex code structures, narrowing down to the locations of where the bugs may happen.

Many automated tools and techniques are designed for bug localization, such as information retrieval, machine learning [15] or SBFL techniques (program spectrum) [22] [23]. These techniques utilizes various type of resources such as bug reports, stack traces or source code files. In the particular case of program spectrum, the tools utilizing this technique rely on a form of ranking metrics or algorithms [22] to rank part of the codes based on the likelihood of bug in that particular location. For example, tools like FLACOCO [21] or GZoltar [4] utilizes code coverage along with test outcomes to assign suspiciousness score to different code components.

Spectrum-Based Fault Localization (SBFL) is a bug localization method that uses dynamic information from test execution [23]. Common techniques used for SBFL tools are metric-based technique. Utilizing code coverage, this technique enables the speculation of suspicious code lines for a better idea on the location of the faulty code. It distinguishes between successful and failing test cases for the assessment of faulty code area. This is a popular technique for bug localization, as the results achieved are remarkable with low overhead costs [22]

Limitations of using SBFL technique in the current settings. Due to the low maturity of SBFL tools, the tools are still not widely available in the public. According to a study conducted by C. Parnin and A. Orso [16] Developers who are experienced with debugging and have to work with simpler code are more likely to better utilize the full capability of the bug localization tool. As developers debug faulty programs with the support SBFL tools, only a small number of them can demonstrate their effectiveness.

Another investigation examined by Higor A. de Souza et al. [22, p.19] reported back three factors against the adoption of SBFL tools in the real world, these include the confusing long lists of suspiciousness. Since the list of suspiciousness may not actually reflect the actual location of the fault in some cases –

which is demonstrated in the results of Chapter 4, the long lists of suspiciousness may even be useless to developers. So SBFL tools should avoid excessive information that can overload users. The second factor is the limited information on why that particular region of code are marked as suspicious. Many developers who participated in the study had reported that they would like to see more information other than the suspiciousness list regarding the variables and test cases related to the suspiciousness element. The third factor is that many developers who participated in the same study have little to no experience in using automated testing and debugging. The reason for this is many software companies do not utilize automated testing, which contributes to the possibility of low adoption of SBFL techniques.

In many cases, SBFL tools have been used to evaluate a set of known real world programs, but they are simple and contain only a small and known amount of bugs. In reality, developers have to deal with more complex programs with an unknown amount of bugs. Therefore, tools for SBFL have difficulty to reach software company debugging process.

The effectiveness of SBFL tools through evaluation. SBFL tools have been applied in the purpose of advancing the field of software debugging by offering mechanisms to identify the location of the bug in the software. The effectiveness of these tools not only impacts the efficiency of the debugging process, but also influences the trust of developers in the automated debugging process.

Many studies have been conducted to evaluate the effectiveness of SBFL tools. In a study carried out by Higor A. de Souza et al. to understand how developers use SBFL technique as well as their effectiveness on locating bugs [23]. The amount of developers aided by SBFL tools are more likely to reach the faulty region of the code. Although the study shows that the correlation of efficiency between developers who used the SFBL tool for debugging and those who do not is insignificant [22, p.11-12]. The feedback from developers who participated in the study was positive regarding the perceived usefulness, ease of use and their likelihood of continuing to use these tools in the future.

As depicted in **Figure 2.5** and **Figure 2.6**, the perceived usefulness and the perceived ease of used recorded from the study with twenty-six participants [23] is divided in multiple categories, evaluated using the Technology Acceptance Model (TAM) [5]. The TAM items are defined in the following nomenclature:

Perceived Usefulness
- **U1:** Using SBFL tool *improves* my performance in the debugging task.
- **U2:** Using SBFL tool in the debugging task increase my *productivity*.
- **U3:** Using SBFL tool enhance my *effectiveness* in the debugging task.
- **U4:** Overall, I find the SBFL tool *useful* to perform the debugging task.

Perceived Ease of Use
- **E5:** *Learning* to use the SBFL tool is easy for me.
- **E6:** Interacting with the SBFL tool does not require a lot of *mental effor*t.
- **E7:** Overall, I find the SBFL tool easy to use.
- **E8:** I find it easy to get the SBFL tool to do what I want it to do.

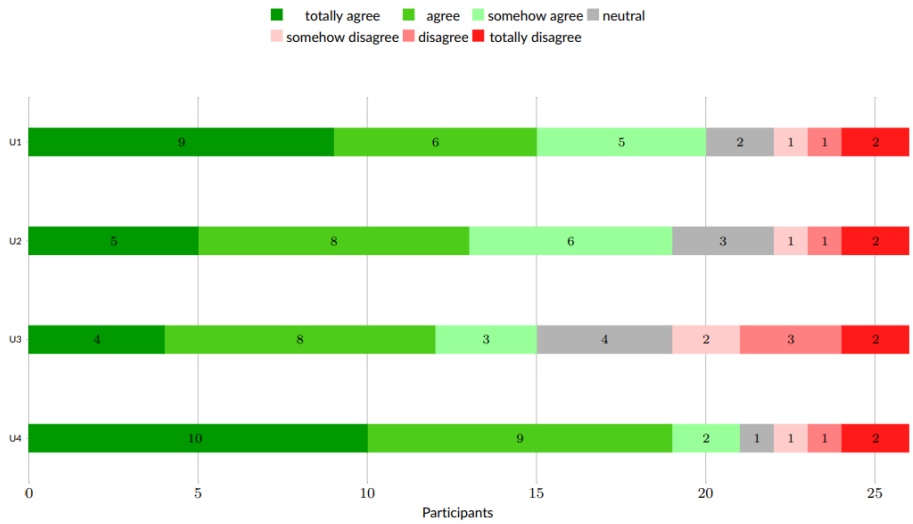


Fig. 2.5. Perceived usefulness [23]

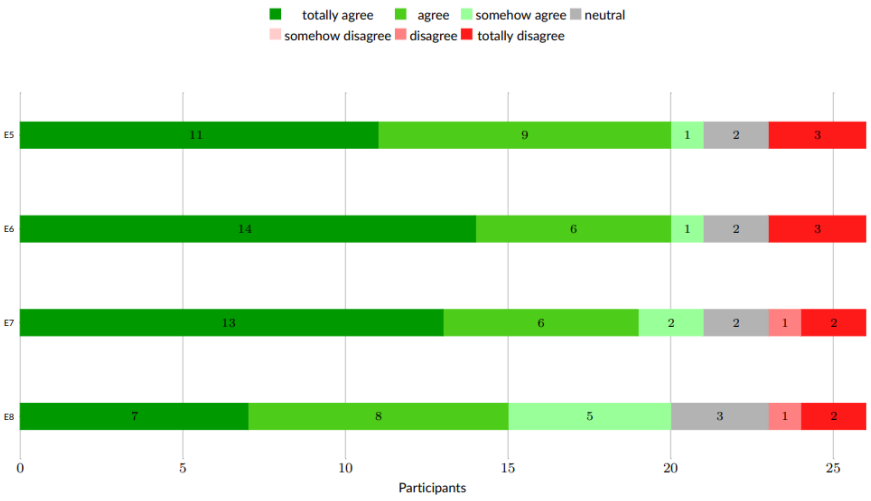


Fig. 2.6. Ease of use [23]

Spectrum-Based Fault Localization techniques offers a wide range of strategies to identify faulty regions of code. Most SBFL tools utilize metric-based techniques, using a form of ranking metrics to separate suspicious regions. Other techniques are statistical-based techniques, execution models, Artificial Intelligence-based techniques, and program dependence-based techniques [22]. There are also other forms of techniques which are combinations of the previously mentioned. However, the metric-based techniques will be the main focus due to their popularity and adaptability in SBFL tools [15].

Metric-based techniques utilize ranking metric formulas to identify potentially faulty components within a program. Many SBFL approaches either use or suggest these metrics to improve the bug localization process. These ranking metrics rely on program spectrum information, which is gathered from testing, to establish correlations between program entities and test case outcomes. The entities are then assigned for each a suspicious score based on how likely it is to be faulty. Therefore, the frequency in which entities are executed in failing and passing test cases is analyzed to calculate its suspiciousness score. Many ranking metrics have been developed specifically for bug localization only, while others were adapted from different fields, such as molecular biology. [22] Some of the ranking metrics shown in **Table 1** were proposed in various studies, which were summarized in a survey carried out by Higor A.de Souza et al. [22, p.8]

Table 1. Ranking metrics and formulas [22]

Ranking metrics	**Formula**	**Ranking metrics**	**Formula**
Tarantula	$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{ep}}{c_{ep}+c_{np}}}$	Ochiai	$\frac{c_{ef}}{\sqrt{(c_{ef}+c_{nf})(c_{ef}+c_{ep})}}$
Jaccard	$\frac{c_{ef}}{c_{ef}+c_{nf}+c_{ep}}$	Zoltar	$\frac{c_{ef}}{c_{ef}+c_{nf}+c_{ep}+10000\times\frac{c_{nf}\times c_{ep}}{c_{ef}}}$
O^p	$c_{ef}-\frac{c_{ep}}{c_{ep}+c_{np}+1}$	O	-1 if $c_{nf}>0$, otherwise c_{np}
Kulczynski2	$\frac{1}{2}\left(\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{ef}}{c_{ef}+c_{ep}}\right)$	DStar	$\frac{c_{ef}^{*}}{c_{nf}+c_{ep}}$

In **Table 1**, the classification for the formula of the ranking metrics are defined as follows: represents the number of times where a program entity (c) is executed (e) during failing test cases (f), while represents the number of times a program entity is not executed (n) in failing test cases. The metric specifies the number of times that a program entity is executed in passing test cases (p),

while specifies the number of times that a program entity is not executed in passing test cases.

The Ochiai ranking metric for fault localization used in SBFL techniques is well-regarded due to its effectiveness in identifying bug locations in a program. A research in bug localization [18] has demonstrate the effectiveness of Ochiai on real faults, as well as shown that the bug localization tool using this formula can successfully pinpoint defects by examining how often program entities are executed in passing versus failing test cases. This approach is especially common due to how it systematically ranks program entities based on their likelihood of being faulty, which aids developers in locating and addressing defects

2.3 FLACOCO: A bug localization tool for Java.

From the drawbacks of current SBFL tools for practical applications to the strength of bug localization techniques and SBFL's potential for practical applications, FLACOCO addresses some of the key challenges faced by other SBFL tools.

FLACOCO, a bug localization tool for Java, significantly enhance the practical aspects of SBFL tools by leveraging industry-grade coverage. Built on top of a community-maintained code coverage library for Java called JaCoCo, the design ensures support for a wide range of upcoming versions of Java, as the library evolves. This makes FLACOCO highly flexible for a variety of localization requirements,

The design of FLACOCO specifically addresses some of the challenges faced by other SBFL tools. While most SBFL tools only work on specific versions of Java such as Java 8, FLACOCO offers itself as the only bug localization tool working on different versions of Java, including the LTS Java 17. Other design decisions that answers some of the problems that other SBFL tools still face include handling non-terminating processes such as infinite loops through isolated test executions and reducing classpath conflicts. [21]

In a research carried out by André Silva et al. [21], to evaluate the support for two bug localization tools which are GZoltar and FLACOCO on different versions of Java LTS, the dataset taken from Defect4J V2.0 [11] – which contains 835 bugs – has been utilized for the experiment. 191 bugs are compatible with four Java LTS versions and has been tested using the two bug localization tools for comparison. The details from **Table 2** shows that FLACOCO have successfully output 125 bugs for all four Java LTS versions with a config, whereas GZoltar's performance varies through each version. In the case of Java 17, GZoltar does not support bug localization entirely.

The difference in performance was statistically considerable across all evaluated target versions. The study [21] emphasizes FLACOCO's capabilities to support multiple Java versions, attributing the performance advantage to JaCoCo for its robustness, making FLACOCO an effective tool for bug localization.

The inner workings of FLACOCO is described in Section 3.2, which outlines the basic bug localization process of FLACOCO and the integration of the service into the LCA system in detail.

Table 2. The comparison for execution of both FLACOCO and GZoltar on 191 bugs [21]

	Fault Localization	
	GZoltar v1.7.3	**FLACOCO v1.0.5**
Defects4J Config	108	125
Java 7	111	125
Java 8	110	125
Java 11	121	125
Java 17	0	125
Total	**450**	**625**

3 Methodology

This chapter is dedicated to explaining the methodology behind the reconstruction of the LCA system, as well as the process of integrating of a bug localization service that is FLACOCO to output bug localization report for any student submission with at least a valid test case.

3.1 Reconstructing the Learner Code Analyzer system

The process of rebuilding the LCA system involves many steps, with the purpose of integrating a bug localization service as a form of feedback, which contributes to enhancing the learning experience. The process of rebuilding the system includes:

- Reviewing the system architecture with the overall application flow
- Setting up the system on a local environment
- Evaluating the state of the system for bug localization integration

System Overview Looking at **Figure 2.1**, the 3 components of the LCA system was designed to support each other for a seamless process of analyzing and generating questions. Further inspection into the analyzer engine, this particular component of the system was designed with the purpose of adapting for futures development, as explained for the design choice of adding multiple customized analyzer [19]. However, the possibility for further developing the analyzer engine does not end here. Utilizing Spring Boot framework with Maven dependencies, the opportunity of developing another service into the current architecture, integrated into the analyzer service is plausible.

However, due to the conflicts in dependency, or more specifically, node dependencies from the front-end of the system, the user interface was not available to user at the start. Other than that, some sections of code relating the web page routing configuration [19, p.37] was also missing, which leads to a number of navigation problems in the webpage. Another issue to address is the missing component from the user interface that requires code reviewing to be fixed. Some

of the back-end issues were also discovered and addressed, which are discussed in Section 3.1

After inspecting and testing out the LCA system after the restoration, a summarized flow of the application in the case of assigning questions to students and submitting submission to the system from the students with the analyzed questions retrieved from the analyzer engine can be described as follows:

- A teacher create assignments.
- Assignments are distributed to students.
- Students submit their work to the system.
- The analyzer engine processes submissions and generate questions based on the analysis.
- Teachers assign these questions back to students for discussions
- After the discussions were made, the assignment can be closed and finalized.

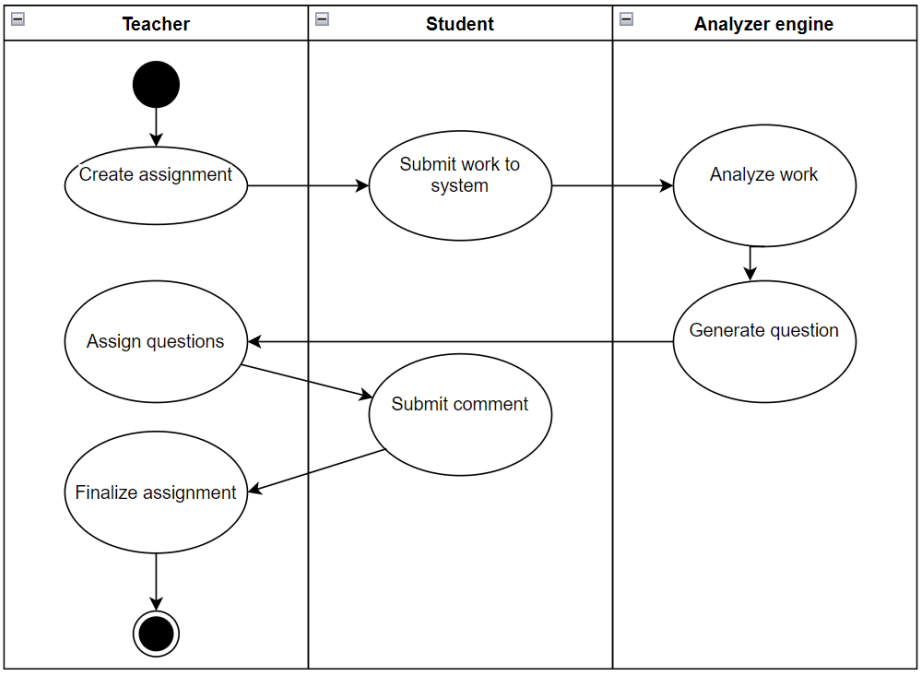


Fig. 3.1. The process of interaction for assignments and generated questions

Setting up the LCA system in a local environment. The original LCA system was developed and tested on an Apple Silicon M1 Pro SoC [19]. The initial state of the LCA system was filled with bugs, acting as a setback for

the testing process as well as the bug localization service integration process. The LCA system was tested on Windows. As shown in **Figure 2.1** the system uses two ORMs, which is Prisma and Hibernate, this has caused multiple issues in the process of debugging for reconstruction of the LCA system, because the initial state of the system stills contains incompletions in some sections of the code. Some of issues referred are mismatching data types between the back-end API and the analyzer when interacting with the database, or the conflicting Maven dependency in the local settings. Performing database generation with DDL generation utilizing Spring Data JPA/Hibernate along with the capability of Prisma for database prototyping [24] using the CLI for interactions between components had met setbacks as the data type cannot be resolved with the current state of the LCA system component's configuration, so a new approach for the schema on testing and debugging must be made for avoid future configuration issues. Other issues to address is the missing routing configuration on the user interface side of the LCA system, along with some missing components such as MUI. The front-end also faced dependency issues, which is important to the interface's functionality. However, the issues were addressed by updating the necessary dependencies and analyzing other sections of the project for clues on how to resolve some structural errors.

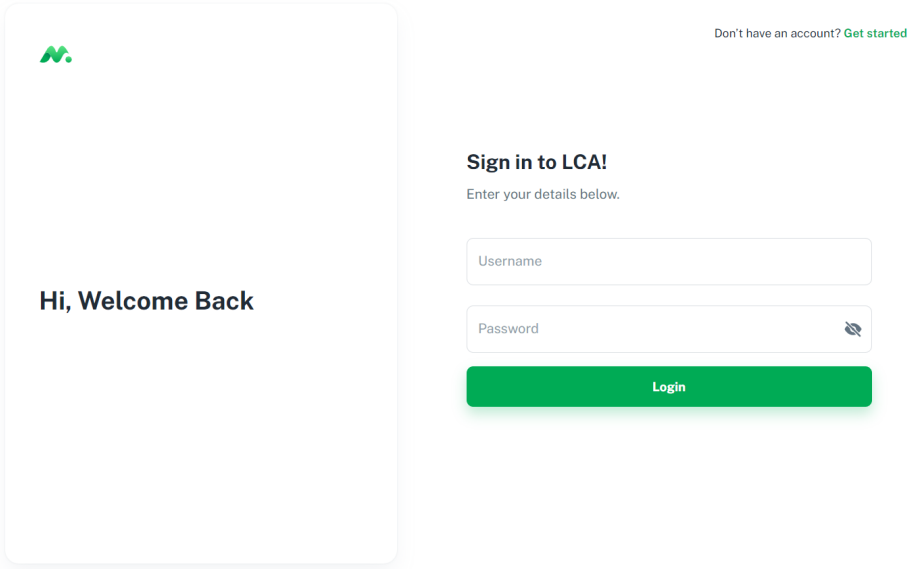


Fig. 3.2. The restored user interface of the LCA system

The section below describes the technology used for the rebuilding of the LCA system in a local environment.

Docker Desktop Utilizing Docker [10], the process of setting up the database was simplified. Since the project was using a PostgreSQL database, Docker allows the user to set up the database using pre-configured images that include the database environment, for this instance, setting up a PostgreSQL image *toberemoved [3]. In this particular case, using Docker allows the testing of multiple databases, not exclusive to PostgreSQL, providing a large variety of option to examine many projects individually without having to install multiple databases and eliminate the need for manual database configuration. Because the upcoming examined projects using the LCA system may contain different types of database configuration, having multiple options to examine other databases might become useful. Because Docker only provides configuration options through the Command-line Interface (CLI), using Docker Desktop can significantly reduce the effort spent on consulting various sources for a way to configure images, with the help of a well-crafted user interface.

DBeaver Configuring multiple databases without having an actual way to interact and monitor them besides the CLI may prove difficult, this is why the method of configuring the databases also includes DBeaver as the DBMS for managing a variety of databases. DBeaver is a universal database management tool to work with data professionally. Some of the features that DBeaver offers are good UI designs, the support of cloud data sources, support for enterprise security standard, multiplatform support, etc. [2] With DBeaver, the LCA system's database can be view thoroughly with the option of creating a mock database as a test environment, dedicating to resolving the issue mentioned above.

By combining the use of the DBMS DBeaver, and Docker as a platform for containing database images, debugging and resolving some of the issues in the LCA system was achievable. Including the mismatching data types for between the database model, using two ORMs, which contributes to a lot of problems happening to the analyzer engine services; the Spring Data repository with faulty configuration, and the mismatching dependencies in Maven configuration.

Evaluation for integration of bug localization service. Overall, the restoration of the LCA system was a success with major issues having already been addressed. However, there may be some underlying features which has not been fully recovered, because of its insignificance to the true purpose of the restoration of the system, the implementation of a bug localization service on the LCA system for the purpose of evaluating its practical application. A feature that has not been restored is the Kubernetes cronjob [19, p. 20] of the analyzer engine. Originally, the analyzer engine was supposed to be deployed on the Kubernetes and scheduled to be run hourly for assignment submissions check. However, the reason for the omission of this functionality is the design decision for the bug localization service does not affect the impact of the presence of the cronjob, the design for bug localization is discussed in Section 3.2. Since the absence of cronjob does not affect the overall system, time and effort can be focused on introducing a bug localization service more efficiently, and any plan for rein-

troducing or replacing the cronjob feature is also viable, as this can eliminate any potential design flaws relating to the Kubernetes cronjob with its current implementation in the future.

3.2 Bug Localization Service Integration

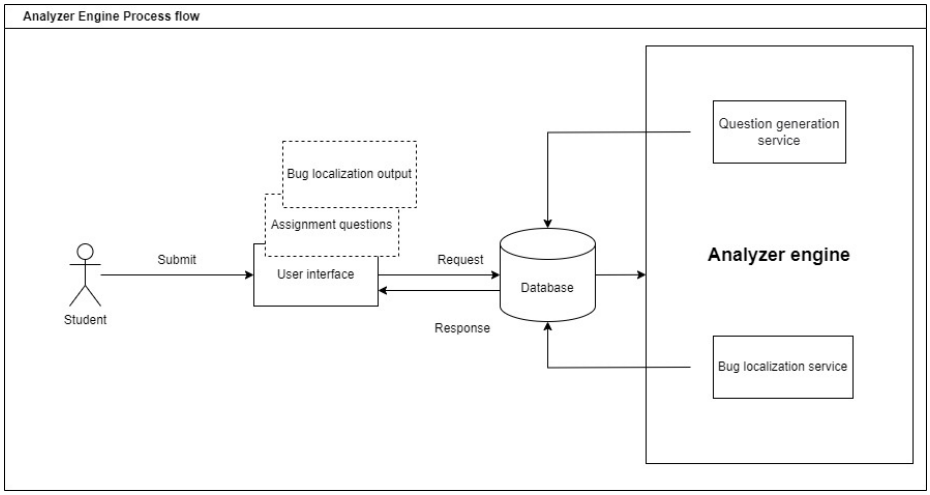


Fig. 3.3. A process flow of generating questions, incorporated with a fault localization service of the analyzer engine

In the analysis phase of the LCA system workflow, which can be seen in **Figure 3.1**, the analyzer engine hosts an amount of time for analyzing the submission from the students, then when the analyzing process is done, a CT model is taken as an input for the custom analyzer templates, which is defined by the user, more specifically, the administrator of the system or the teacher. In order to elaborate the design decision, the analyzer engine will be separated into two services. The question generation service, and the bug localization service. Details for the question generation service can be referred to Section 2.1 for the full process. In the next sub-section, the design bug localization tool, FLACOCO will be further illustrated for the bug localization service integration.

FLACOCO's design As illustrated in **Figure 3.4**, the design of FLACOCO consists of four steps for bug localization, which is Test Detection, Instrumentation and Test Execution, Coverage collection and Suspiciousness Computation. [21]

– **Test Detection**: In this process, FLACOCO identifies and analyzes test cases by scanning for the compiled program (in Java bytecode) and filtering

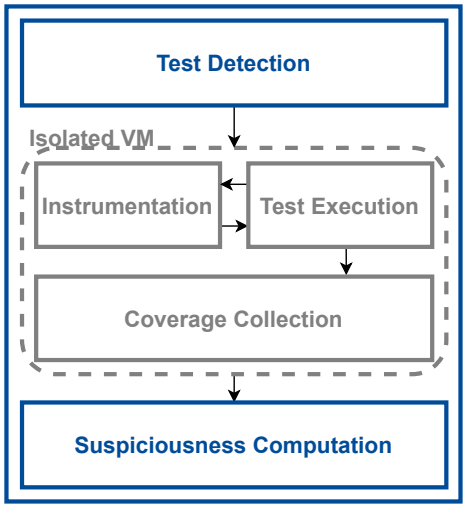


Fig. 3.4. An overview of the inner workings of FLACOCO [21]

those that match the specifications of the supported test framework. Currently FLACOCO can support JUnit 3, JUnit 4 and JUnit 5, which is a wide range of test drivers.

- **Instrumentation and Test Execution**: To clarify, code instrumentation is the process of injecting code into existing program to output data during program execution for monitoring (e.g., printing an output of a variable during runtime) [9]. Here, FLACOCO execute all of the identified tests in a separate process, and this execution is monitored by a custom Java agent for the purpose of recording the lines covered by test cases. Using 'test-runner,' a library for executing JUnit test, the instrumentation process for classes during test execution occurs at the time of class loading and is entirely facilitated by the Java code coverage library JACOCO, which is recognized for its industry-grade capabilities.
- **Coverage collection**: FLACOCO gathers data at the end of each test, including the test outcomes, any exceptions thrown, and line-by-line coverage for each class tested.
- **Suspiciousness**: The conclusion of the test suite execution gives out the results for each test execution that enables the computation of the suspiciousness score of each covered line. The score is determined by which tests covered the line, the outcomes of the tests and the selected formula for score calculation. In default, FLACOCO uses Ochiai for the reasons mentioned at the end of section 2.3. It is worth mentioning that this is not the only formula available in the selections, as Tarantula was recently added to the selections in early 2024. [20]

The source code for FLACOCO can be found here

Adaptability of FLACOCO to the LCA service. With the inner workings of FLACOCO explained, this section is for explaining how the process of integrating the bug localization service comes together. An idea of the general flow of the application can be found in **Figure 3.3** Building on the flow of the application, as well as consulting the original activity diagram in **Figure 3.1** , the analyzer engine can be implemented with a bug localization service as follows: The system initially scans the database for new program submissions stored as repository URLs in distributed version control systems (DVCSs). It then make use of Git commands to clone the program's source code for processing with the Spoon parser, which constructs and manipulates Java source code ASTs. After the cloning process, the bug localization service becomes active and scan for the project to compile both the project and the test cases, which acts as an input for the test detection phase of FLACOCO, described in Section 3.2. After the bug localization process is finished, the system continues to the question generation service. Eventually, the parsed ASTs are analyzed by custom analyzers template pre-defined by users. Each is equipped with specific questions templates and criteria related to the structure of the code. The system allows the user to contribute their own custom analyzer through a defined framework that aligns with the analyzer engine's standards. Finally, both the bug localization report and generated questions are then stored in the database to provide data for interaction between the front-end and the back-end component of the LCA system.

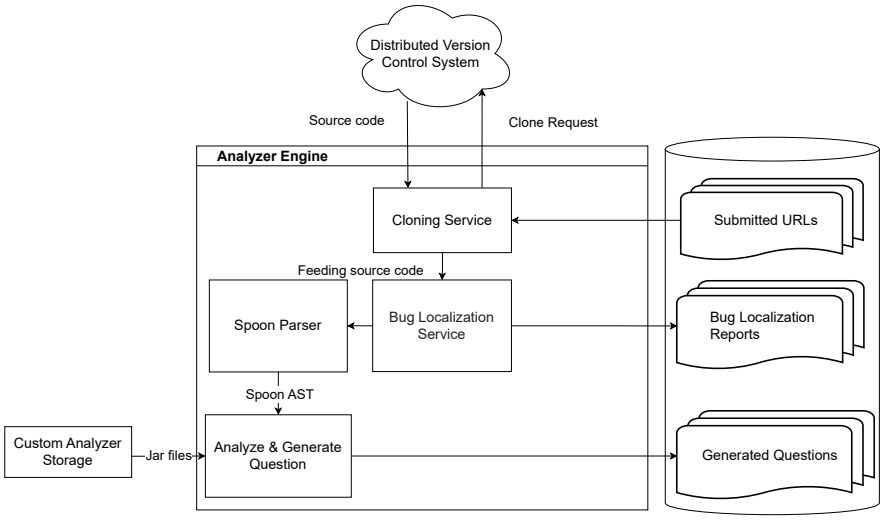


Fig. 3.5. A new general process of the analyzer engine, with the new bug localization service

The integration of the new bug localization report is expected to contribute to the LCA system as a form of feedback for the student in order to increase the learning experience, as well as providing data for any developmental research.

4 Implementation And Results

4.1 Implementation

Previously illustrated in **Figure 2.1**, the technical architecture for the LCA system includes three components. But this chapter will specifically focus on the implementation of the newly implemented bug localization service: FLACOCO and its modifications made on the three component of the analyzer service. The system was rebuilt and tested the on Windows 11, using a 12th Gen Intel(R) Core(TM) i5-12450H processor and 16GB of RAM.

Bug localization service: Analyzer engine. The source code for the new Analyzer Engine can be found here

With guidance from the Java API documentation [20] from FLACOCO [21], integration of the bug localization into the analyzer engine is a straightfoward process, as FLACOCO only contains two main classes, Flacoco and FlacocoConfig.

Listing 4.1. A simple implementation for setting up FLACOCO

```
1 //set config option
2 Flacoco flacoco = new Flacoco(config);
3
4 FlacocoResult result = flacoco.run();
5
6 Set<TestMethod> failingTests = result.getFailingTests();
7 Map<Location, Suspiciousness> mapping = result.
      getDefaultSuspiciousnessMap();
```

For the sake of simplicity, the default mode of FLACOCO will be used for the integration of the bug localization service. The process of setting up Flacoco inside the analyzer engine is a straightforward process, which can be described through the following steps: **a)** Instantiate a 'FlacocoConfig' object for configuration(refers Listing 4.1). **b)** Setting up configuration using the 'FlacocoConfig' object. **c)** Create a new 'Flacoco' instance that passes the specified configuration in step a. **d)** Use the created 'Flacoco' instance to execute the 'run' method, saved in a 'FlacocoResult' object. It is worth noting that the formula used to calculate suspiciousness score in this report is Ochiai.

Originally, there were four classes that provided services for different process (refer to **Figure 2-2**) in the analyzer engine: AnalyzerService, AssignmentService, CloneService and SpoonService. A brief description of the services chronologically:

- AssignmentService contains the full process flow of the analyzer enginer, it serves at the backbone to call other services inside the analyzerAssignment() method.
- CloneService handles the cloning process from a distributed version control system. It is responsible for retrieving the assignment submission of students.
- SpoonService is responsible for parsing the project into ASTs and returning a suitable CT model, ready for question generation.
- AnalyzerService is implemented for the purpose of inserting the returned CT model into a suitable custom analyzer template. The templates use the CT model to generate a series of questions and save it into the database.

In order to setting up FLACOCO inside the analyzer engine of the LCA system, a service for bug localization called 'FaultLocalizationService' has been implemented that allows the incorporation of the bug localization service inside the usual analyzer engine process.

The FaultLocalizationService utilizes the method runFaultLocalizationService(), which take in a student assignment id as a parameter. Whenever the method is called, it checks for the specific id to verify whether the assignment exists in the database. If the assignment is found, it will start the bug localization process and compile the project through the use of a Maven dependency called Maven Invoker [1], which allows the service to run multiple Maven projects programmatically. By searching for a specific 'pom.xml' file in the project recursively (taking the file closest on surface of the searching algorithm), it will be able to recognize the environment of the submitted project and configure the project path, with the compiled target file as an input for FLACOCO. Using these information, FLACOCO will search for compiled test cases, and return a list of executed test cases, failed test cases, the location of the suspicious line for faulty code and the suspicious score for each line. Then, utilizing Spring Data JPA, the output will be saved in the database for back-end and front-end requests and responses.

Since the FaultLocalizationService store the bug localization report in the database, changes to the database must also be made in order to adapt with the newly implemented service. Addressing the use of 2 ORMs in the system, particularly in the back-end and the analyzer engine; the changes made related to the database must also be configure for the ORM of the analyzer engine, which utilizes Hibernate; and the back-end API, which utilizes Prisma.

The FaultLocalizationService outputs a bug localization report, because of the configuration for the process, the report data is divided and stored using two separate entities. The reason for this separation is because to ensure Boyce-Codd normal form that the original database provide [4, pp. 18-19]. The added entities are described in the table below:

Bug localization service: Back-end The source code for the new back-end component of the LCA system is here

A few additional adjustments have been made for the back-end API of the LCA system in order to accommodate the additional bug localization service. As

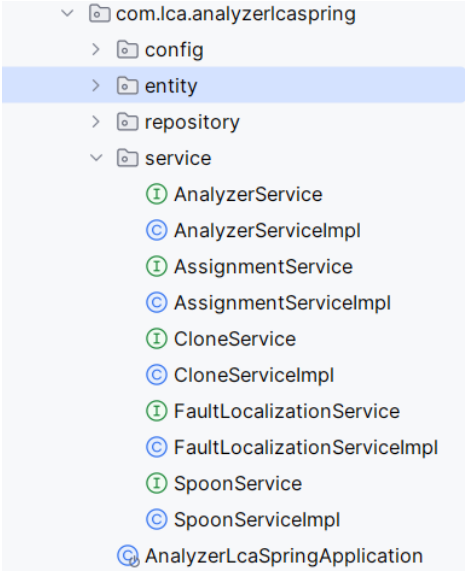


Fig. 4.1. The new implemented structure utilizing a bug localization service

Table 3. The added entities for the bug localization service

Entity	Description
localization_report	This table contains the predicted location of the bug, specifically the class name, the suspiciousness score, and the line number where the bug occurs.
executed_tests	This table - as the name suggests - contains the executed tests, which is a Boolean column to check if that executed test has failed or succeed.

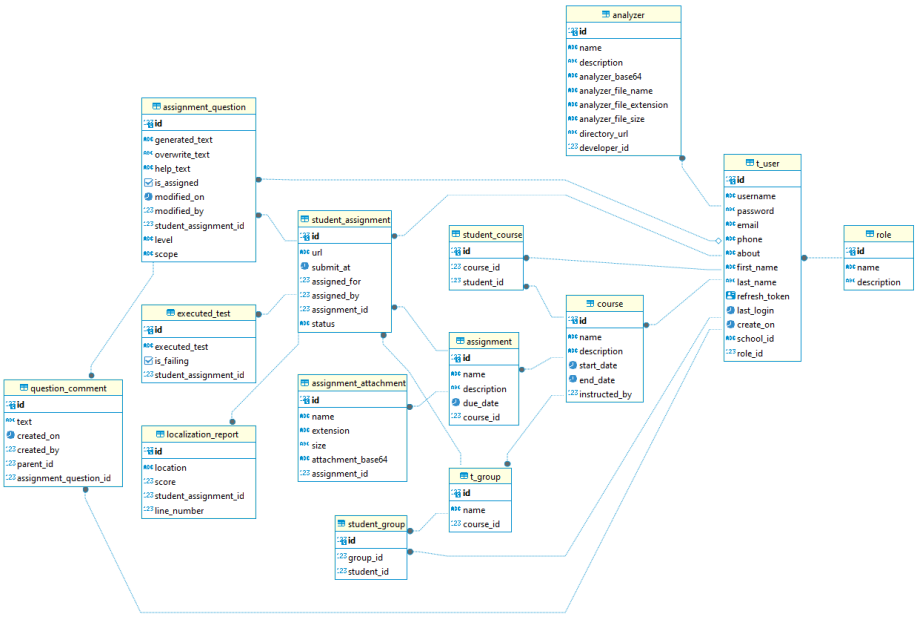


Fig. 4.2. Full Entity-Relationship diagram of the LCA system with the bug localization service

mentioned in **Table 3**, the entities created in the analyzer engine are already in a functional state, the data gathered from the bug localization process have been gathered and saved in the database. However, in order to interact with data and display it on the front-end of the LCA system, modifications to the GraphQL schema must be made.

Figure 4.2 displays the entity relationship diagram of the LCA system with the two implemented entities. Using DBeaver for testing to avoid the data type mismatch between the two ORMs from the analyzer engine and the back-end, creating new GraphQL model is fairly simple using the documentation [6] and referencing the original code. Using GraphQL 'types', which are used to specify the structure and the data types of object within a GraphQL schema – similar to 'entities' used in the analyzer engine, the tables seen in **Figure 4.2** are created, particularly the *executed_test* and the *localization_report*.

Listing 4.2. Creating GraphQL types

```
type LocalizationReport {
  id: Int!
  location: String!
  lineNumber: Int!
  score: Float!
  studentAssignmentId: Int!
  studentAssignment: StudentAssignment
}
```

```
9
10 type ExecutedTest {
11   id: Int!
12   executedTest: String!
13   isFailing: Boolean!
14   studentAssignmentId: Int!
15   studentAssignment: StudentAssignment
16 }
```

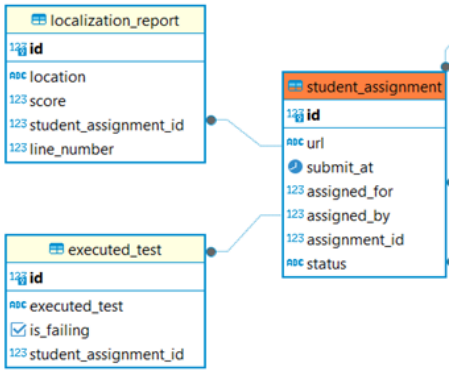


Fig. 4.3. Simplified E-R diagram of the implemented sections

In order to interact with the front-end of the LCA system, modification to the query functions must be made to operate with the types created in **Figure 4.3**. The query functions can be described as following figure:

Listing 4.3. Server-side functions for querying the database for bug localization needs

```
1 async function getAllLocalizationReportsByAssignmentId(parent,
      args, context, info){
2     const {studentAssignmentId} = args;
3
4     const reports = await context.prisma.localizationReport.
          findMany({
5         where:{
6             studentAssignmentId: studentAssignmentId
7         },
8     });
9     if (reports === null) return null;
```

```
    return reports
}

async function getAllExecutedTestsByAssignmentId(parent, args,
    context,info){
    const {studentAssignmentId} = args;

    const tests = await context.prisma.executedTest.findMany({
        where:{
            studentAssignmentId: studentAssignmentId
        },
    });
    if (tests === null) return null;
    return tests
}
```

Both functions make use of Prisma's 'findMany' method for query performing and execution, which is a part of Prisma's client API for interaction with the database.

Bug localization service: Front-end The source code for the Front-end component of the LCA service is available at here

The query function have been implemented in the back end of the system. However, in GraphQL APIs standard, queries serve to fetch data from a server. They allow the freedom to request of specific data format on the client side, which means that several client-side requests can be handled by a single server side query. [19] Because of this, any requests from the front-end can be specified using the query resolver that it needs. Both **Listing 4.3** and **Listing 4.4** illustrate this. Although GraphQL defining a schema in the front-end is not necessary, defining a query request is crucial for using the application, as the query defines exactly what data the front-end can retrieve and use, for the sake of targeted data interaction between client and the server.

Listing 4.4. Client-side query requests for user interface data interaction

```
export const LOCALIZATIONREPORT_BYASSIGNMENTID_QUERY = gql`
    query getAllLocalizationReportsByAssignmentId(
        $studentAssignmentId: Int!) {
        getAllLocalizationReportsByAssignmentId(
            studentAssignmentId: $studentAssignmentId) {
            id
            location
            score
            lineNumber
            studentAssignmentId
        }
    }
`;

```

```
13 export const EXECUTEDTEST_BYASSIGNMENTID_QUERY = gql`
14     query getAllExecutedTestsByAssignmentId(
           $studentAssignmentId: Int!) {
15         getAllExecutedTestsByAssignmentId(studentAssignmentId:
               $studentAssignmentId){
16             id
17             executedTest
18             isFailing
19             studentAssignmentId
20         }
21     }
22 `;
```

With the data flow properly handled, a webpage that illustrates bug localization results is implemented. Refers to **Figure 3.3**, after the bug localization service has finished its process, the bug localization results will be displayed on a user interface. Diving into the details of this process, after a student submit an assignment, the assignment will be marked as 'ASSIGNED' and the status will be stored in the database. After the process of bug localization has been completed, along with the question generation process, the status of the assignment will be marked as 'GENERATED.' During this process, the bug localization page will not appear if the status of the assignment is being assigned to students or being submitted. Instead, it will only appear if the status is from 'GENERATED' to 'CLOSED' as in **Figure 4.4** below.

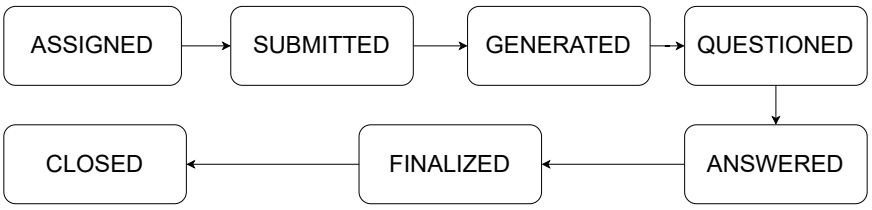


Fig. 4.4. The status of assignment submissions demonstrated in stages

The user interface of the bug localization report includes a table that holds a list of executed tests and an indication of whether that test failed, along a list of localization reports, which comprises of a suspiciousness score for each line, the speculated location of the bug and the line where it is.

4.2 Results

This section shows the results from the implementation of the bug localization service into the LCA system, the results will be separated into two parts, the results from the implementation and the results from the experiment, which will be used for evaluation in Section 5

Implementation results The implementation results are shown from restoration of the LCA system, and the integration of the bug localization service. In detail, this section depicts the actual results and improvements considered from the modification of the LCA system, giving a clear view on the impact of the implementation.

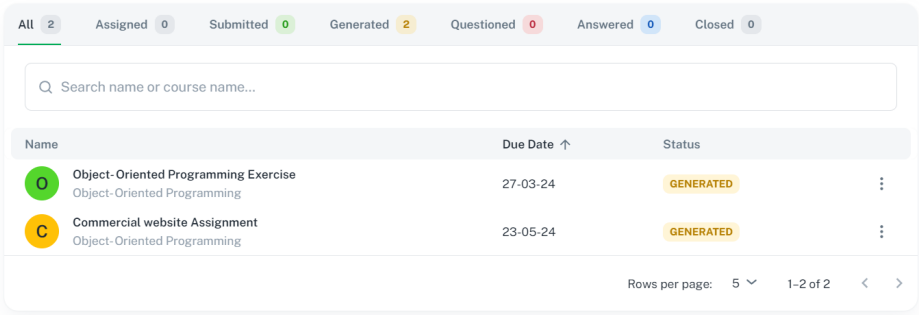


Fig. 4.5. The user interface displaying the assigned assignment page

The restored user interface shows the user list, the course list, the assignment section and the analyzer section, as shown below:

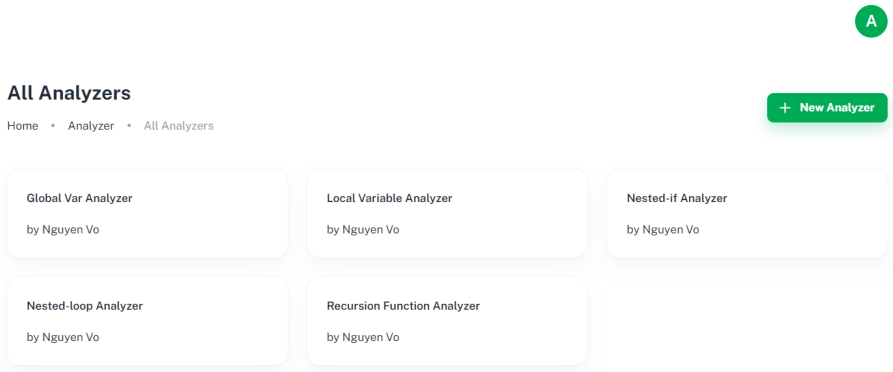


Fig. 4.6. A list of Customized analyzer list for the analyzer engine

With the user interface restored nearly to its original state, the bug localization report interface was also implemented in flow of assignments for students, as described in Section 4.1.

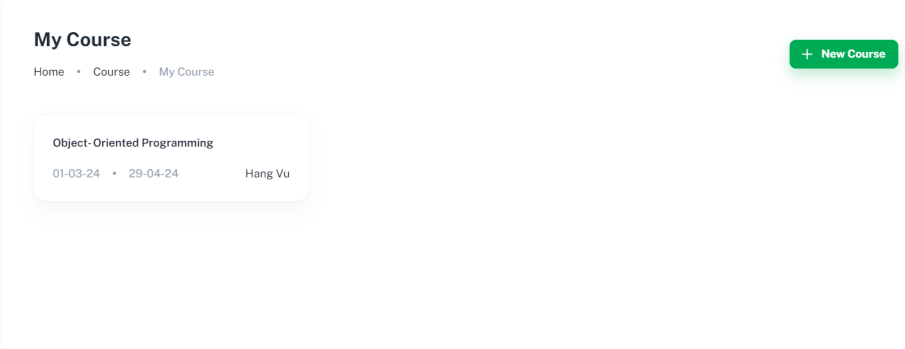


Fig. 4.7. The courses page shown as tabs

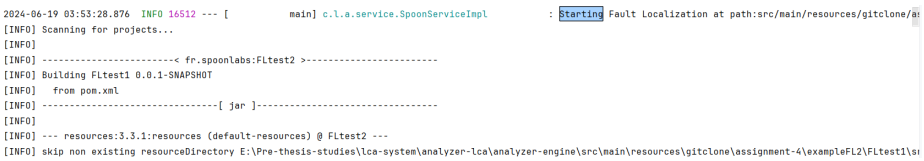


Fig. 4.8. The user page, shown as a list

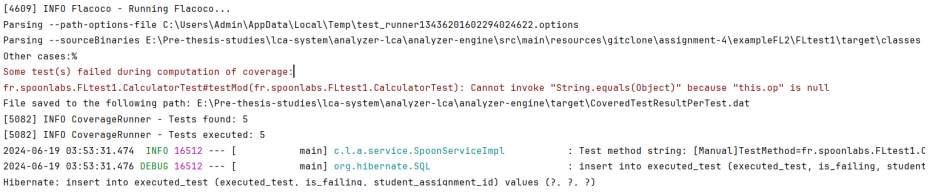


Fig. 4.9. FLACOCO reports on test found and executed to insert the data collected

When the analyzer engine finishes its process, the bug localization report data is collected from the database and shown in the 'assignment' → 'assigned' tab.

In the assigned tab, user can select an assignment that has been set to 'Generated,' select the 'view' option to enter the assignment details page. There, user can see an option for viewing generated question and the bug report. The bug report button enters the page for a complete bug localization report taken from the project analyzed.

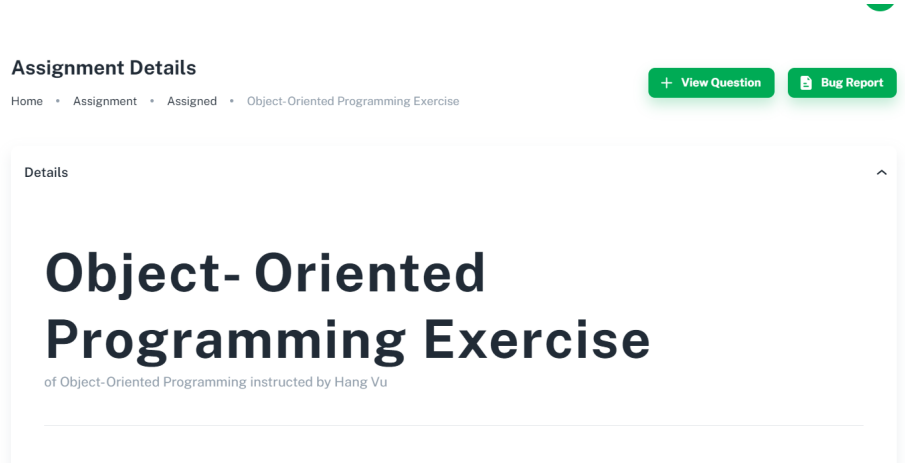


Fig. 4.10. An overview of the assignment details page

As shown in **Figure 4.11** and **Figure 4.12**, the bug localization report contains various information that aid students and teachers as a form of feedback to encourage the discussion between users, which is one of the reasons for the implementation of the bug localization service.

Experimental results This section is dedicated to exploring the performance of the implemented bug localization service through five projects. It is worth mentioning that the tests for these project were written with consideration of the known bugs in these projects. Therefore, all of the bugs in the projects are known bugs. Specifications for the projects are described in the table below:

Using Ochiai (refers to **Table 2.1**) as the main formula for to calculate suspiciousness score, the failing test cases which may or may not execute the code section and the passing test cases which may or may not execute the code plays a major role for evaluation.

Calculator: Firstly, project No.1 is used as a test to verify the functionality of the bug localization service. This project used one of the examples given

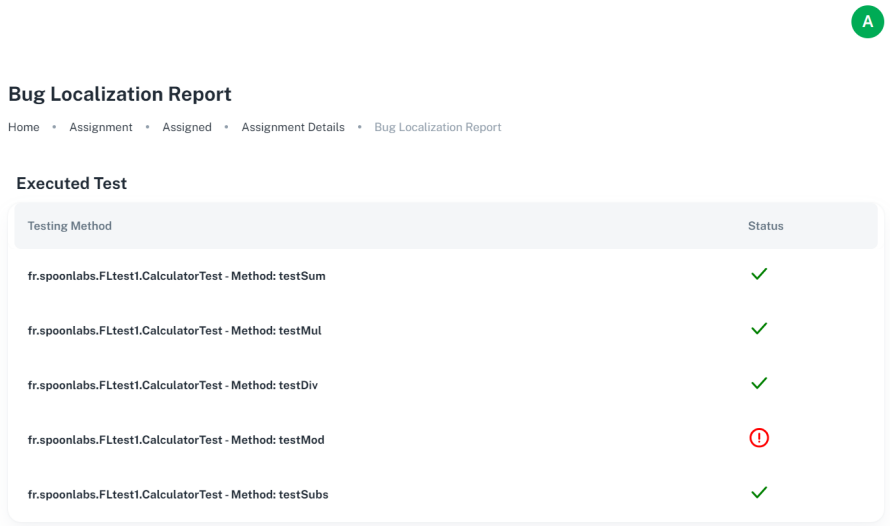


Fig. 4.11. Bug localization report: executed tests and failed tests

Fig. 4.12. Bug Localization main report interface

Table 4. Projects used for experiment on FLACOCO

No.	Project Name	JUnit	Framework	Classification
1	Calculator	4	No	Demo
2	Calculator 2	4	Spring Boot	Demo
3	Calculator 3	5	No	Demo
4	Data Structure and Algorithm Sorting	4	No	Exercise
5	E-Commerce Platform	5	Spring Boot	Real Application

in FLACOCO's Github source code. Five tests was executed in this project, utilizing JUnit 4 as a framework for unit testing. The test methods are testSum, testMul, testDiv, testMod and testSub. The failed test is testMod, due to a Null-Pointer Exception in the code.

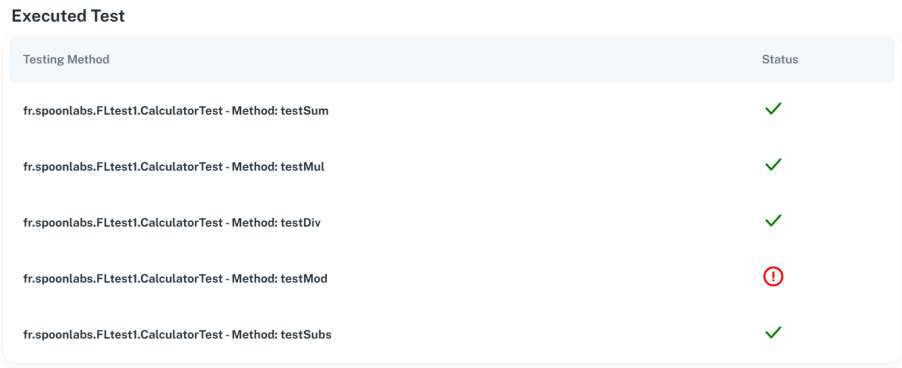


Fig. 4.13. The executed test of project No.1

Listing 4.5. The buggy location in project No.1

```
10 public int calculate(String op, int op1, int op2) {
11
12     if (op.equals("+")) {
13         return op1 + op2;
14     } else if (op.equals("-")) {
15         return op1 - op2;
16     } else if (op.equals("*")) {
17         return op1 * op2;
18     } else if (op.equals("/")) {
19         return op1 / op2;
20     } else {
21         System.out.println("Other cases:" + op.toString());
22         if (this.op.equals("%")) { //NPE
23                 return op1 / op2;
24                 }
```

```
25            }
26        throw new UnsupportedOperationException(op);
27    }
```

Fig. 4.14. The Localization Report for project No.1

Elaborating how the suspiciousness score was calculated using Ochiai, the Ochiai formula suspiciousness score is calculated by taking the number of failing test cases which executed the code (c_ef), divided to the square root of the sum of c_ef and the number of failing test cases which did not execute the code (c_nf) times the sum of c_ef and the number of passing test cases that executed the code (c_ep).

In the particular case of project No.1, the faulty code in **Listing 4.5** is the line 22, $c_ef = 1$ as the failing test case which executed the code is *testMod*, $c_nf = 0$ as there are no failing test case which did not execute the code. $c_ep = 0$ as there are no passing test case the executed the code.

$$\text{suspiciousness}_{22} = \frac{c_{ef}}{\sqrt{(c_{ef} + c_{nf})(c_{ef} + c_{ep})}} = \frac{1}{\sqrt{(1+0)(1+0)}} = 1 \quad (1)$$

Examine another case in line 18, because the key for applying Ochiai is knowing which lines are executed by failing and passing test case, I will be using the data from the code coverage library JaCoCo to uncover the value of c_ef, c_ep and c_nf, we have the following value: $c_ef = 1$: indicating there is one failing test case where the line is executed $c_ep = 1$: indicating there is one passing test case where the line is executed $c_nf = 0$: indicating there are zero failing test case where the line is not executed

$$\text{suspiciousness}_{18} = \frac{c_{ef}}{\sqrt{(c_{ef} + c_{nf})(c_{ef} + c_{ep})}} = \frac{1}{\sqrt{(1+0)(1+1)}} = \frac{1}{\sqrt{2}} \approx 0.7071 \tag{2}$$

This aligns with the data shown in Figure 4.14. Overall, the bug localization service works as expected with Project No.1.

Calculator 2: Project No.2 is used to verify the capacity of FLACOCO when working with frameworks, specifically Spring Boot and FLACOCO's capacity of detecting tests involving the functionality of the framework. In this particular experiment, the capability of Spring involving the MVC architecture is also evaluated.

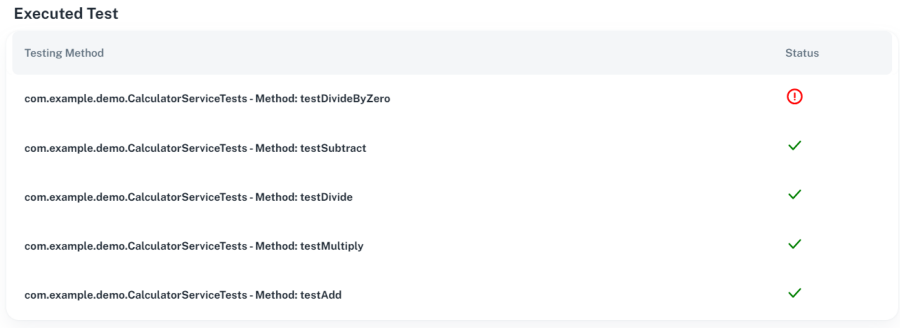


Fig. 4.15. The executed test for project No.2

Fig. 4.16. The localization report for project No.2

In this experiment setup, there are 6 tests written for the project to check the basic functions of the calculator class. However, from the results in **Figure 4.15**, there are only five tests executed, this is because the sixth test is a MVC-related test, shown in **Listing 4.6**. It verifies the ability to call an API defined in class CalculatorController for arithmetic operations. Upon further inspection, the usage of MockMvc in the test may be the cause why FLACOCO cannot detect

this test. Upon speculation, it may be because of the import path of MockMvc which is related to 'org.springframework.test.*' that may not be accessible to FLACOCO.

Listing 4.6. The test that was not found by FLACOCO

```
public class CalculatorControllerTests {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private CalculatorService calculatorService;

    @Test
    public void testAddition() throws Exception {
        double a = 10.0, b = 15.0;
        when(calculatorService.add(a, b)).thenReturn(25.0);
        mockMvc.perform (get("/add")
                        .param("a", String.valueOf(a))
                        .param("b", String.valueOf(b))
                        .contentType(MediaType.
                            APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.
                    APPLICATION_JSON))
                .andExpect(jsonPath("$", closeTo(25.0, 0.001))
                    );
    }
}
```

Calculator 3: This experiment section will be about verifying the ability of the bug localization service to operate on a different JUnit version. Using an example prepared from FLACOCO's GitHub source code, project No.3 was taken as an assignment submission. The results can be found in the **Figure 4.17** below:

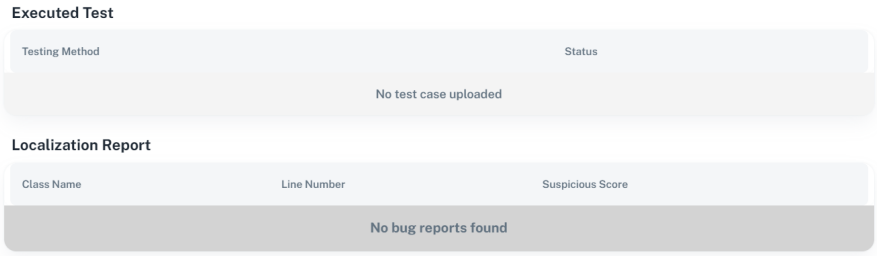


Fig. 4.17. Bug Localization report of Project No.3

There are 5 tests, which are similar to project No.1. However, the test cases written was not found by the bug localization service. The reason for this is because the bug localization service is having a problem configuring the setting to detect JUnit5 tests. However, testing JUnit5 projects through the CLI using the similar setting as the Java API through FLACOCO v1.0.7-SNAPSHOT.jar does not encounter any issue. This might be because of the bug localization service that was implemented on FLACOCO v1.0.6, the configuration for JUnit5 may encounter some issue, preventing the bug localization service from being utilized.

Data Structure and Algorithm Sorting: Project No.4 was taken and modified from an actual programming exercise in a Data Structure and Algorithm course. By modifying this project into using Maven with a test suite, this project was used to verify the bug localization service capability in a real-life scenario, specifically when using in classes. The full bug localization report can be found in the figures below: Project No.4 was set up with a faulty implementation of two

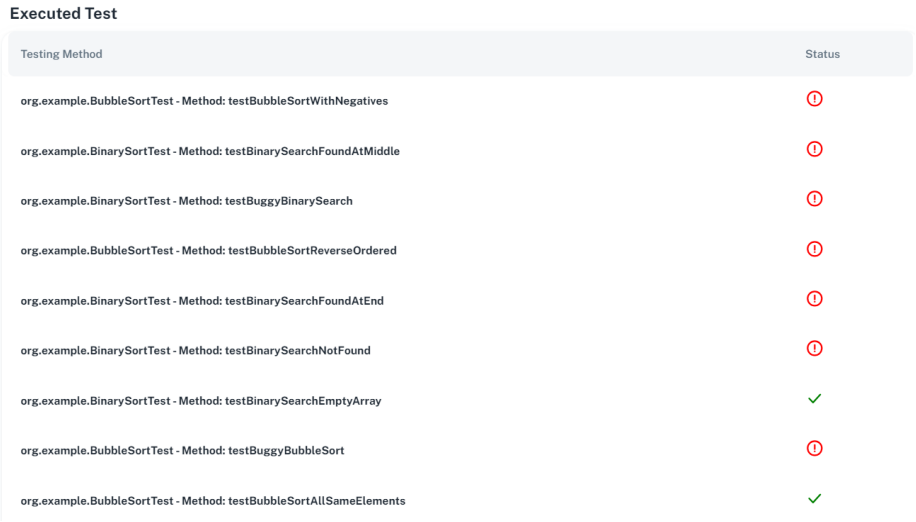
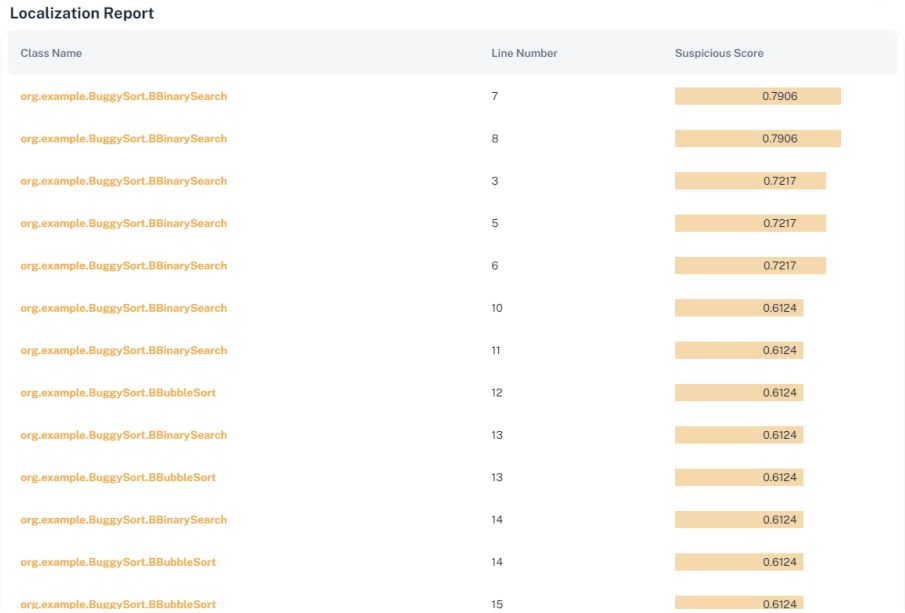


Fig. 4.18. The executed test for project No.4

sorting method: the bubble sort and a binary search. An observation can be made that, in **Figure 4.19**, the list of the localization report is quite long, even though the speculated locations reported in the figure does not necessarily mean that the error is there, but rather, the specific region of the error is narrowed. Each method contains a test suite which comprises of six test cases. According to **Figure 4.19**, the BBinarySearch class contains the fault at line 7 and 8 with an equal suspiciousness score of approximately 0.7906, the BBubbleSort contains the fault at line 12 with a suspiciousness score of 0.6124.



Localization Report

Class Name	Line Number	Suspicious Score
org.example.BuggySort.BBinarySearch	7	0.7906
org.example.BuggySort.BBinarySearch	8	0.7906
org.example.BuggySort.BBinarySearch	3	0.7217
org.example.BuggySort.BBinarySearch	5	0.7217
org.example.BuggySort.BBinarySearch	6	0.7217
org.example.BuggySort.BBinarySearch	10	0.6124
org.example.BuggySort.BBinarySearch	11	0.6124
org.example.BuggySort.BBubbleSort	12	0.6124
org.example.BuggySort.BBinarySearch	13	0.6124
org.example.BuggySort.BBubbleSort	13	0.6124
org.example.BuggySort.BBinarySearch	14	0.6124
org.example.BuggySort.BBubbleSort	14	0.6124
org.example.BuggySort.BBubbleSort	15	0.6124

Fig. 4.19. The localization report for project No.4

Listing 4.7. Location of the bug in the bubble sort method

```
3  public class BBubbleSort {
4      public void bubbleSort(int[] arr) {
5          int n = arr.length;
6          boolean swapped;
7          for (int i = 0; i < n - 1; i++) {
8              swapped = false;
9              for (int j = 0; j < n - i - 1; j++) {
10                 if (arr[j] > arr[j + 1]) {
11
12                     int temp = arr[i]; //bug: should be arr[j]
13                     arr[j] = arr[j + 1];
14                     arr[j + 1] = temp;
15                     swapped = true;
16                 }
17             }
18             if (!swapped)
19                 break;
20         }
21     }
22 }
```

Listing 4.8. Location of the bug in the binarySearch method

```
3  public class BBinarySearch {
4      public int binarySearch(int[] arr, int x) {
5          int l = 0, r = arr.length - 1;
6          while (l <= r) {
7              int m = l + (r - l) / 2;
8              if (arr[m] > x) // bug: should be arr[m] == x
9                  return m;
10             if (arr[m] < x)
11                 l = m + 1;
12             else
13                 r = m - 1;
14         }
15         return -1;
16     }
17
18 }
```

Overall, the results is positive in this experiment, as it successfully demonstrates the capability of the bug localization service in coding assignments.

E-commerce platform: Finally, Project No.5 is a web-based project for the purpose of online shopping. This is the back-end of an E-commerce platform system, which contains various classes and methods from utilizing RESTful API to building services and repository in a MVC architecture. As expected, since the project

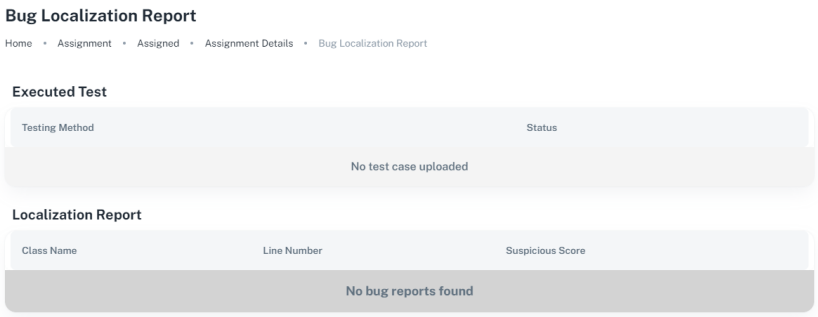


Fig. 4.20. The bug localization report of project No.5

is utilizing Spring Boot on a big scale, and the tests are written using JUnit 5, the bug localization service cannot detect and execute the tests. For reasons mentioned in experiment of project No.3 and project No.2. Running the project in FLACOCO v1.0.7-SNAPSHOT using the CLI, the result is still the same, as shown in **Figure 4.21** The conclusion drawn from this experiment: Due to the

```
2024-06-19 03:53:26.358  INFO 16512 --- [           main] c.l.a.service.SpoonServiceImpl           : Maven compilation success
[0] INFO Flacoco - Running Flacoco...
[17] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.AddressController
[19] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.CustomerController
[20] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.LoginController
[22] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.ProductController
[23] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.ShoppingCartController
[36] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.service.configuration.CorsConfig
[37] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.service.configuration.SecurityConfig
[39] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.service.FileStorageService
[42] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.service.implementation.CustomerServiceImpl
[45] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.service.implementation.ProductServiceImpl
[46] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.service.implementation.ShoppingCartServiceImpl
[53] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.AddressController
[54] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.CustomerController
[54] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.LoginController
[55] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.ProductController
[55] WARN TestMethodFilter - NoClassDefFoundError: class com.project.eplatform.controller.ShoppingCartController
```

Fig. 4.21. The thrown error of project No.5

low maturity of FLACOCO, the bug localization service is not suitable for big projects, utilizing different components of a framework which FLACOCO may not have supported yet in test coverage.

The source for 5 projects can be found here:
Project No.1
Project No.2
Project No.3
Project No.4
Project No.5

5 Discussion And Evaluation

This chapter delves into the detailed discussion from the results in Chapter 4, obtained from the integration of FLACOCO into the LCA system.

5.1 Discussion

The experiment above provide insights on the use of FLACOCO and the practical application that it provides, as well as the limitations that it still has.

- Project No.1 served as an initial test to validate the basic functionality of FLACOCO bug localization service. The experiment has proven to be successful and validates its purpose.
- Project No.2 targeted the service's compatibility with frameworks, specifically Spring Boot. Even with the successful identification of some bugs, challenges can be observed when the service have to deal with tests related to MVC components. The experiment suggests that while FLACOCO can be adapted to framework based application, it stills have difficulties adapting to bigger scale projects due to low maturity of FLACOCO.

- Project No.3 reveals the current state of FLACOCO being unable to use JUnit 5 API-wise. Instead, JUnit 5 projects can only be used using a snapshot version of FLACOCO, which has not been released as a public dependency.
- Project No.4 provides information on FLACOCO's performance on algorithmically related assignments. The project's focus on data structure and algorithms tested FLACOCO's analysis capability in identifying faulty codes. The results was promising, demonstrating FLACOCO's potential in educational setting where detailed feedback on algorithms are important.
- Project No.5, the experiment with a large scale E-commerce platform outlines the limitations of FLACOCO in large-scale applications.

From the results of the experiment, it is clear that the implementation of FLACOCO as a bug localization service still has room for development. The inability to detect and support various frameworks for testing environment is still an issue that many bug localization programs still have to face. Identifying which tests to run, and then execute the tests is a difficult task, and this is a problem for many bug localization and code coverage tool, not exclusive to FLACOCO [22].

The performance of FLACOCO in practice has also demonstrated a major point of bug localization, which is the difference between syntax error and logical error and the purpose of bug localization. To summarize, syntax error happens when the program does not follow the rule of the programming language, while logical error is when the program does not follow the expected behavior of the programmer, or usually known as software bug. Most of the time, syntax error can easily be fixed as it is detected by the compiler as soon as it happens; detecting logical error is not so straightforward and usually relies on test cases most of the time. Since the problem with test cases is that it does not offer much information to work with, having a bug localization service can narrow down the location of the code in a large codebase, which can potentially optimize the workflow of developers.

5.2 Evaluation

The bug localization service has met all of the basic requirements to use as a service integrated into the LCA system. One particular improvement for this service is that while it can run perfectly on Maven projects, other projects template such as Ant or Gradle are not supported, which can be an inconvenience. Some form of implementation for supporting other project templates have been made but has been dropped due to the time it will take for further testing on these project templates.

However, FLACOCO is a great bug localization tool with the potential to be integrated industry-wise by the capability to work on multiple Java LTS versions. The experiment shown in 4.2 collectively demonstrate FLACOCO holds substantial promise for educational and small-scale applications. By examination from the experimental results section above, a suggestion table for courses that can use Java as the main language for programming and utilize bug localization in situations such as lab assignments or small exercises, is shown in **Table 5**.

In order for FLACOCO to be viable in larger range of applications, particularly in educational technology where the learning outcomes can be enhanced, improvements must be made to focus on the configuration flexibility as well as supporting newer frameworks.

Table 5. Courses and Bug Localization Feasibility

Course (Java)	**Feasibility**	**Explanation**
Data Structures and Algorithms	Yes	Experiment on project No.4 shows that using a data structures and algorithm assignment can effectively localize bugs.
Object-Oriented Programming	Yes	Results from project No.2 and No.4 demonstrate that focusing on the algorithmic aspect of OOP can verify bug localization using test cases.
Web Application Development	No	Project No.5, and partly project No.2, demonstrate that bug localization in a complex programming environment remains challenging for the tools used.

6 Conclusion And Future Work

6.1 Conclusion

In conclusion, this thesis offers the integration of a bug localization service into the LCA system. The implementation of the bug localization service realizes a potential impact on education practices with a detailed feedback that the service provide, helping teachers and students to have more data – particularly about programming errors - to expand the discussion meaningfully, as well as giving teachers ideas to come up with questions.

The LCA system still have some flaws, such as the outdated dependencies in the analyzer engine that can be remedied by refactoring codes along as well as reviewing logical correction. However, the system works as expected, and it is expandable with the presence of a bug localization service.

6.2 Future work

Mentioned in the previous chapter, the bug localization service along with the LCA system still has room for improvement. Other than the fact that the technical structure of the system can be reviewed for enhancement, improvements related to the features of the system can be described as follows:

– **Expansion to support multiple language:** Supporting languages other than Java can be a huge feature which can help teachers to expand their

range of teaching. However, the main difficulty of this proposal is the approach to cross-language implementation. The current bug localization service is tailored for Java. Taking this approach means finding a suitable way to refactor the code structure to adapt multiple language, and different sets of bug localization tool that support other languages. A suggested direction for this approach is to implement a bug localization hub that can take in multiple bug localizations services for different languages, and then adjust the original workflow of the LCA system to incorporate the bug localization hub seamlessly.

– **Extending Bug Localization using other techniques:** Other the SBFL, testing bug localization with other techniques such as information retrieval techniques or machine learning and implementing them with multiple options can be beneficial to students and researchers by interacting with a large data of assignment submission

References

1. Apache maven invoker. `https://maven.apache.org/shared/maven-invoker/` (2024), [Accessed: 16 June 2024]
2. Dbeaver documentation. `https://dbeaver.com/docs/dbeaver/` (2024), [Accessed: 13 June 2024]
3. baeldung: Postgresql with docker setup. `https://www.baeldung.com/ops/postgresql-docker-setup` (2024), [Accessed: 13 June 2024]
4. Campos, J., Riboira, A., Perez, A., Abreu, R.: Gzoltar: An eclipse plug-in for testing and debugging. 2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings (09 2012). `https://doi.org/10.1145/2351676.2351752`
5. Davis, F., Davis, F.: Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Quarterly **13**, 319– (09 1989). `https://doi.org/10.2307/249008`
6. Facebook, Inc: Graphql. `https://graphql.org/learn/` (2024), [Accessed: 12 June 2024]
7. Gierke, O., Darimont, T., Strobl, C., Paluch, M., Bryant, J., Turnquist, G.: Spring data jpa. `https://docs.spring.io/spring-data/jpa/reference/index.html` (2024), [Accessed: 12 June 2024]
8. Gupta, R., Kanade, A., Shevade, S.K.: Deep learning for bug-localization in student programs. ArXiv **abs/1905.12454** (2019)
9. Ilas, F.: Using code instrumentation for debugging and constraint checking (2009)
10. Inc., D.: Docker docs. `https://docs.docker.com` (2024), [Accessed: 12 June 2024]
11. Just, R., Jalali, D., Ernst, M.: Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: International Symposium on Software Testing and Analysis (07 2014). `https://doi.org/10.1145/2610384.2628055`
12. Kusmiyati, K.: Analysis of independence and creative thinking skills of students through assignment. Jurnal Pijar Mipa **17**, 754–758 (11 2022). `https://doi.org/10.29303/jpm.v17i6.4232`
13. Martinez, M., Monperrus, M.: Astor: A program repair library for java. In: Proceedings of ISSTA. Saarbrücken, Germany (2016)

14. Matthews, K., Janicki, T., He, L., Patterson, L.: Implementation of an automated grading system with an adaptive learning component to affect student feedback and response time. Journal of Information Systems **23**, 71–84 (06 2012)
15. Mohsen, A., Hassan, H., Moawad, R., Makady, S.: A review on software bug localization techniques using a motivational example. International Journal of Advanced Computer Science and Applications **13** (01 2022). https://doi.org/10.14569/IJACSA.2022.0130231
16. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: 2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings (07 2011). https://doi.org/10.1145/2001420.2001445
17. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: A library for implementing analyses and transformations of java source code. Software: Practice and Experience **46** (08 2015). https://doi.org/10.1002/spe.2346
18. Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 609–620 (2017). https://doi.org/10.1109/ICSE.2017.62
19. Quang, C.K.: Automatically Generating Questions About Code Of Students With Abstract Syntax Tree. Bacherlor's thesis, International University, School of Computer Science and Engineering (2022)
20. Silva, A., et al.: Flacoco wiki. https://github.com/ASSERT-KTH/flacoco/wiki (2023), [Accessed: 2024]
21. Silva, A., Martinez, M., Danglot, B., Ginelli, D., Monperrus, M.: Flacoco: Fault localization for java based on industry-grade coverage (2023), https://arxiv.org/abs/2111.12513
22. de Souza, H.A., Chaim, M.L., Kon, F.: Spectrum-based software fault localization: A survey of techniques, advances, and challenges (2017), https://arxiv.org/abs/1607.04347
23. de Souza, H.A., de Souza Lauretto, M., Kon, F., et al: Understanding the use of spectrum-based fault localization. Journal of Software: Evolution and Process (November 2022). https://doi.org/10.22541/au.166756977.70529286/v1
24. Team, P.: Prisma documentation. https://www.prisma.io/docs/orm (2024), [Accessed: 12 June 2024]
25. Thakker, N.: Scaler. https://www.scaler.com/topics/global-variable-in-java/ (April 2024), [Accessed: 2 June 2024]