An Exploration of Maven-Based Java Repositories and Their Testing Practices

Canol Simsek
Department of Computer Engineering
Izmir Institute of Technology
Izmir, Turkiye
email: canolsimsekk@gmail.com

Tugkan Tuglular
Department of Computer Engineering
Izmir Institute of Technology
Izmir, Turkiye
email: tugkantuglular@iyte.edu.tr

***Abstract*—With the increasing significance of testing in software development, particularly in gaming and e-commerce, these industries continue to thrive and evolve, ensuring that the software systems powering them are robust, reliable, and capable of delivering an exceptional user experience. The research aims to compare the testing practices of two GitHub fields and automate repository mining, test code scanning, and gathering source code metric processes. The study aims to uncover common testing usage compared to class and method counts in the gaming and e-commerce software repositories. This exploration provides valuable insights regarding overall test coverage on repositories and how test usage affects code quality. An automated tool is developed for repository mining that clones repositories from desired topics. The main objective of this project is to gather the test usage data and source code sizes by creating and using static source code analysis tools to answer if the test usage in terms of test classes and test methods changes by the sizes of the repositories and does testing have a negative correlation with code smells. Our findings align with our test usage and size metrics expectations.**

Keywords-unit testing; repository mining; e-commerce software; game software.

I. INTRODUCTION

The project is motivated to provide insights into developers' knowledge of testing used in industries, how other industries handle testing compared to theirs, and guide how to improve testing practices and determine if these topics can be automated. With the increasing significance of testing in software development, particularly in gaming and e-commerce, these industries continue to thrive and evolve, ensuring that the software systems powering them are robust, reliable, and capable of delivering an exceptional user experience. Testing plays a pivotal role in achieving these objectives by detecting and addressing potential issues, enhancing system performance, and safeguarding against vulnerabilities. By comparing the testing practices of gaming and e-commerce repositories, valuable insights can be gained into the similarities, differences, and overall effectiveness of testing approaches in these domains. This knowledge will benefit developers by providing them with guidance on improving their testing methodologies and empowering stakeholders to make informed decisions regarding software quality assurance. Ultimately, the findings of this project aim to contribute to the continuous improvement of software quality.

This project aims to mine public repositories, analyze their source codes, and test usages. An automated tool, which clones repositories from desired topics, is developed for repository mining. The main objective of this project is to gather the test usage data and source code sizes by creating and using static source code analysis tools. Collected metric data is processed to create tables and graphs to compare the two GitHub topics: gaming and e-commerce. These visuals provide insights into these questions:

- How does the test usage change with the size of the repositories in terms of test classes and overall classes?
- How does the test usage change with the size of the repositories in terms of the test methods and overall methods?

Our approach for these topics is creating repository miner software to clone public repositories of desired topics from GitHub. After that, create and utilize static source code analysis tools that will scan cloned repositories for their test class count, test method count, class count, and method count. Cloned repositories will be uploaded to SonarQube for analysis and inspection by their code metrics. The repositories successfully uploaded to SonarQube are then scanned by our test code analysis tool, which will search the repository for test classes and methods. Following these steps, each project result from SonarQube and our analysis tool will be combined and used to create tables. From these tables, visuals will be created to gain insights about our topics.

The paper is organized as follows: Section II presents the related work. Section III explains the proposed approach. Section IV presents the result and discussion, and the last section concludes the paper.

II. RELATED WORK

CORVIG [1] created a pioneering study about patch coverage. The paper helps to create a well-developed code review with a highly efficient approach. Its main objective is to spot possible errors and bugs in the systems. With this paper, we learn how to create such infrastructure and a deeper understanding of code analysis. This paper helped us to understand both code inspection infrastructure and to create a tracker tool for our repositories [1].

In [2], Hassan et al. presented a brief history of the Mining Software Repositories (MSR) field and showed guidance about recent methods for pinpointing the bugs, deployment logs, archived communications, or essential aspects of repositories. Then, they discussed the opportunities for what can be performed to improve this field [2].

Williams et al. [3] studied code coverage with their evolution and provided a tool for bug findings on source code. They describe a method to use the source code change history for refining them and searching for bugs. With the bug database that developers and users of the applications can contribute, they applied mining source code materials and checked over them for bug findings. A static source code checker has done this process. The results are more effective using historical data than other static scanners. They indicated that this project needs to expand, not relying on the user and developer bug reports; projects can provide a new set of bug record data with new types of bugs [3].

Gousios and Spinellis [4] provided a valuable study about how data obtained from GitHub is unsuitable for every research aspect and how this data can be used in large-scale projects. They have used a quarriable offline mirror of GitHub API data for this project to pinpoint the pitfall avoidance strategies. They showcased a GitHub API for streaming metadata from repositories for writing, managing, and optimizing complex queries [4].

Cosentino et al. [5] conducted a meta-analysis of 93 research papers on how they handled data mining from GitHub. The research addressed three dimensions of those 93 papers and addressed poor sampling techniques, lack of longitudinal studies, and replicability issues. Improvements can be made to these topics by developer's data and solution sharing and researchers comparing their results with each other. They can also make guides to how they can clone their projects to make comparisons. They believed these steps could make a general change in confidence in GitHub data [5].

When searching or creating a database in GitHub, researchers had problems with GitHub limitations. Sampling Projects in GitHub for MSR Studies (Dabic et al. [6] provided a GitHub Search dataset with 735,669 repositories to address these issues. With GitHub's millions of repositories, a research paper addressed how much of this data is useful. The systems combine many selection criteria to get the most valuable combinations on GitHub [6].

Kalliamvakou et al. [7] studied quality and available data on GitHub. They analyzed how users handle GitHub features and pointed out the difference between actual data and mined data. They pointed out that maybe the biggest problem for data validity is bias to personal use. Nearly 40% of all pull requests do not appear as merged, even though they were, and half of the users do not have public activity. The paper provides recommendations for developers about how to approach the data on GitHub. As a rule of thumb, the best way to identify a project's activeness is to look at its pulls and commits requests, and the committers are more significant than two [7].

Chaturvedi et al. [8] retrieved all the papers from 2004 to 2013 about Mining Software Repositories published in ICSE. They have analyzed the papers that contained experimental tools or techniques for data mining and repository mining. They have categorized the tools used in MSR on the topics of newly developed, traditional data mining tools, prototype states, and current scripts [8].

By the topic of co-evolution, software needs to evolve, or it will become less valuable over time. Studying the co-evolution of production and test code, Zaidman et al. [9] provided three views that combine information from change history, growth history, and test coverage evolution reports. They applied these three views to two open-source projects and one industrial case to make observations. With these points of view, developers can define different co-evolution scenarios. They also indicated that mining a version control system will provide insight into the testing process [9].

In 2005, Mierle et al. [10] assembled over 200 second-year undergraduate repositories. They have implemented a complete system that parses repositories into an SQL database. The paper examined these repositories' student behaviors, code quality, and code metrics by examining individuals working on the same project separately. However, they point out that the performance indicators cannot predict grade performance. Their results suggest that students' habits and code quality have little effect on their performance [10].

Arcuri and Yao [11] introduced a new view to the co-evolution of software programs and test development. Their approach is to competitive evolution so that both software and testing should directly affect each other. Thus, they co-evolve like prey and predators in nature. The framework is based on co-evolution and search-based software testing [11].

Zaidman et al. [12] studied co-evolution to create awareness among developers and managers alike about the following testing process. In the paper named 'Mining Software Repositories to Study Co-Evolution of Production & Test Code,' they have investigated whether production code and the accompanying tests co-evolve by exploring a project's versioning system, code coverage reports, and size metrics and evaluated their results with the help of log-messages and the developers of the systems [12].

Yalçın [13] developed a co-evolution tracker tool for software with acceptance criteria. The thesis contained 21 real-world projects. Projects are analyzed for every updated and co-evolution process that has been documented. They indicated that when considering Semantic Versioning, Major and Minor version updates have a better ratio for test updates. However, for the result, they found out that even considering major and minor updates, the test update to all update count ratio is not always close to 1.0 [13].

With the evolving complexity of software and test methods, a Literature Review on Software Testing Techniques by Jamil et al. [14] discussed the existing and future testing techniques and how they can be more efficient and enchanted. The paper aims to guide developers to understand and develop their current understanding of software testing techniques for both pre- and post-development cycles [14].

Our work differs from above literature in concentrating on two specific domains and compares the projects with top test usage to explore any patterns.

III. Proposed Approach

The proposed approach is composed of three steps:

1. Cloning GitHub repositories
2. Scanning GitHub repositories for tests
3. Analysis of GitHub repositories with SonarQube
4. Data analysis

The first three steps are explained in detail in this section. The fourth step is presented in the following section with results and discussion.

A. Cloning GitHub Repositories

Repositories are cloned from GitHub by a repository mining tool. This tool works by sending GET requests with GitHub API. This tool loops through GitHub search pages and projects to decide if the projects provide the requested aspects. If these aspects are provided, it clones them to the local library and creates a folder for them. Here is the example API search line:

'https://api.github.com/search/repositories?q=topic:gaming+language:java&sort=stars&order=desc&per_page=100&page=1'

'/search/repositories': Base URL for the repository search. Provides access to GitHub functionalities. The query component of this URL, denoted by '?q=topic: gaming+language: java': is searching for repositories that are tagged in gaming topic ('topic: gaming') and written in Java ('language: Java'). The parameter '&sort=stars' shows the results are arranged based on the number of stars they've received, which is a good indicator of the popularity among the GitHub. In addition, '&order=desc' makes sure these sorted results are presented in a descending order, meaning repositories with the highest number of stars appear first. The number of results can be changed by changing the 'per_page' parameter.

In this research, our tool will look at the top one hundred repositories of desired pages to efficiently collect repositories. GitHub will limit the request per second if the process is anonymous. This limit will slow down the search process. To bypass GitHub limitations, a user token is needed. The user can get this token from authentication settings in GitHub. The tool then sends a Get request and retrieves a JSON response. It will be cleaned if the repository name has an invalid character that may cause problems. However, not every search result will be cloned. To efficiently scan the repositories, third-party programs will be in use. These programs will work best for Maven-based projects. These projects will have a characteristic file named pom.xml. The tool will search for the pom.xml file in the repositories. This search is performed by checking the 'get_contents' method provided by the GitHub package. The 'pomCheck' method will take two parameters ('parent_directory and 'repo'). 'list' is a list of repositories from search parameters. It will loop through every repository on the API calls. 'Repo' is an instance of the 'repository' class from the 'GitHub' Package. It refers to the repository for the method that will be examined.

If the search string 'havepom = repo.get_contents(path ='pom.xml')' returns true with no exception, this means the repository has the pom.xml file. The reason behind checking the pom.xml file is that our tool analyzes only Java projects that used Maven as its build tool. Pom.xml is a Maven-specific Project Object Model (POM) XML file that contains project layout and configuration information. After the project passes the pom.xml file check, the repository will be cloned, and a folder will be created by its name in the local directory. This process will loop through searched all repositories. With 100 repositories for each page, the program will take some time to clone all projects. So, it will also count the repositories for the user and give an indicator on the console so that the user will have knowledge about what part of the search process they are in at that current time.

After all the process is completed, there will be olders depending on the user selection of the topics. In this research, there are two topics: Gaming and E-commerce.

B. Scanning GitHub Repositories for Tests

Our static source code analysis tool has been used to get repositories' test usage metrics. The tool is designed to analyze Maven projects to gather test usage data.

The 'os' and 'glob' packages form the backbone for this tool as they have been used to navigate the folders and operating systems and efficiently retrieve files based on specified path patterns. To ensure accurate reading of the file content regardless of the encoding, the 'FileUtils' class uses the 'charset' package. As a universal encoding detector, 'charset' identifies the encoding of a file, thus enabling 'FileUtils' to read the file content without errors.

On the other hand, the 'javalang' package is a critical component for two classes: 'TestClassCounter' and 'TestMethodCounter'. This package, targeted at parsing Java 8 source code, helps convert the code into an Abstract Syntax Tree (AST), reflecting the hierarchical structure of the source code. The 'TestClassCounter' class uses 'javalang' to parse code into AST and iterates through it to count class declarations. If an error occurs, it ensures a graceful fallback, returning 0 and an empty list. The 'glob' package gets files from pathnames or specified file path patterns. It is used to find all test files from the provided path. The 'pandas' package is a data processing package that stores the results from this analysis. The tool contains five classes. The 'Main' class contains the logic of execution. The 'FileUtils' class reads the file content and encoding detection. The 'RepoParser' class finds test files in the given repository. 'TestClassCounter' is used for counting the test class counts in a given Java repository. This class parses the given code into an Abstract Syntax Tree using the 'javalang' package and then iterates through it to find class declarations. If it gives an error, it returns 0 and an empty list.' TestMethodCounter' is like the 'TestClassCounter' class, which counts method counts in each repository. It also provides total lines of code in the test methods.

The 'Main' class initializes a list of directories to be scanned. After the initialization, it looks for git repositories using the 'get_repos()' method from the 'RepoParser' class. It then finds all the repository's test files, opens them, and checks their encodings using 'chart.' The 'FileUtils' class reads the test codes, and 'TestClassCounter' with ''TestMethodCounter' is called. With these findings, the database is created with methods with their method names, classes that methods are located, files that classes contained, folder paths that these files contained, and each repository with their total test lines of codes.

C. *Analysis of GitHub Repositories with SonarQube*

SonarQube is an open-source platform [15]. It provides continuing code quality management with static code analysis. It can detect bugs, code smells, security issues, and overall code quality. The reason it has been selected for this research is that it can give us valuable metrics about repository sizes so that these can be compared to understand the relationships between them.

SonarQube can be run on different operating systems. The user must select the operating system they are currently using and run the 'sonar.bat' file in the folder. The program will set up its configurations and launch a local server. By default, it will be 'localhost:9000'. The SonarQube server has the following aspects: A web server that serves as the user interface, a search server that utilizes Elasticsearch, a computation engine for code analysis reports, and a database for storing code metrics and instance configurations. It is helpful to note that, in some instances, there might be multiple Java versions on the user's server. In this case, the user needs to define what Java version will be used in the search process. Instead, it is also possible by setting the environmental path for Java to 'SONAR_JAVA_PATH' in the user's local path settings.

Once the analysis is performed, repository metrics can be received from SonarQube. For this, different automation tools have been developed. This tool is also written in Python and uses 'requests' and 'pandas' packages. It communicates with SonarQube and collects code quality metrics. The desired metrics can be selected, or all can be included. After that, it creates an Excel file to store them. With the SonarQube authentications like previous usage, users must have a URL, Username, and Password. Then, the tool will send Get requests with authentication to SonarQube API to retrieve metric data. It then processes the JSON response to extract the metrics by their domain. It then creates a dictionary named 'metric_domains' where each metric key is a new domain. It then iterates over all repositories. For each repository, it creates a 'project_metrics' dictionary. The results are gathered from JSON and stored in that dictionary. After these steps, the tool opens 'ExcelWriter' with the 'XlsxWriter' engine. Then, for each domain that is keyed, it creates a new sheet in the file.

In our research, the following metrics are evaluated:

• Class count: Number of classes written in the project (without test classes).

• Test Class Count: Number of test classes written in the project.

• Method count: Number of methods written in the project (without test methods).

• Test Method Count: Number of test methods written in the project.

IV. Results and Discussion

With the repository mining, source code scanning, and retrieving data from API finished, metrics tables are created. The goal is to compare the two topics' test usage metrics to their sizes and code quality metrics to their test usages. Linear regression, scatter plots, and cluster graphs is used for these tasks. These methods help us to visualize these topics. Intuitive correlations between the metrics are as follows: A positive relationship is expected between class counts and test class counts. Test usage should also be expanded as codebases expand to reduce bugs, errors, code smells, and unwanted program behavior. A positive relationship is also expected with method counts and test method counts. The reason is the same as the class/test class comparison. Test usage is essential for code development and expansion.

Metrics about gaming software repositories can be seen in Table 1. Table 1 lists class count, test class count, method count, and test method count for the five projects with the highest test class counts. Although the ezyfox-server project has the highest number of tests classes among the projects under consideration, it has comparatively less test method than the base project. The base project has the highest number of test methods. The Open Realm of Stars project has the highest test class-test method ratio. Class counts and test class counts would be expected to be similar in optimal cases. However, for the projects in Table 1, there is barely any correlation. Test usage, i.e., test class count and test method count, can be highly different from one repository to another. The same observation can be made for method and test method behaviors.

TABLE I. TOTAL CLASSES/METHODS AND TEST CLASSES/METHODS FOR GAMING SOFTWARE

GitHub Projects	*Class count*	*Test class count*	*Method count*	*Test method count*
ezyfox-server	663	337	2553	943
base	785	314	3742	4256
Open Realm of Stars	302	190	3471	1245
jeveassets	1024	76	8987	743
mmo	184	14	1633	44

The same metrics are collected for the five e-commerce software projects with the highest test class counts and presented in Table 2. Although the io.spot project has the highest number of test classes and of test methods, there are only 11 test classes and 25 test methods. The IOM Project Bootstrap Archetype project has the highest class count-test class count ratio, whereas the productsv project has the highest method count-method class count ratio. The test usage rate seems independent of repository size. There are repositories that have thousands of methods and still have less than 25 test methods.

TABLE II. TOTAL CLASSES/METHODS AND TEST CLASSES/METHODS FOR E-COMMERCE SOFTWARE

GitHub Projects	*Class count*	*Test class count*	*Method count*	*Test method count*
io.spot-next:spot-framework	430	11	2025	25
microservices-event-sourcing/parent	203	9	479	16
IOM Test Framework	337	6	1667	20
productsv	27	6	73	12
[Tool] IOM Project Bootstrap Archetype	16	4	56	6

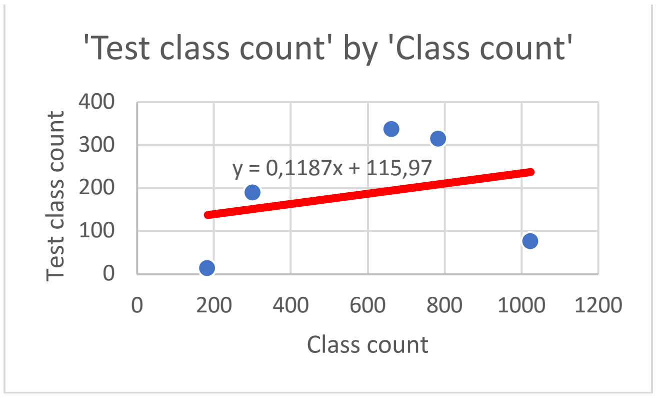

Figure 1. Test Class Count by Total Class Count for Gaming Topic.

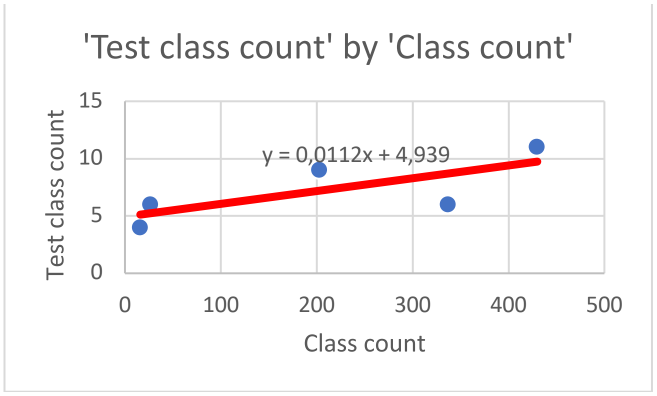

Figure 2. Test Class Count by Total Class Count for E-Cmmerce Topic.

Figure 1 and Figure 2 represent scatter plots of the highest five projects for gaming and e-commerce software projects, respectively. Gaming software projects have higher class and test class counts and higher test class/class ratios. The game software developers tend to write more tests than the e-commerce software developers. Still, both domains need to improve their test development effort.

This project aimed to find answers to these two questions:

- How does the test usage change with the size of the repositories in terms of test classes and overall classes? Answer: Every Java repository has different characteristics. With the Gaming and E-Commerce topics, test usage changes from topic to topic and even between projects. While gaming repositories are much larger than e-commerce repositories, their test usage is unsatisfactory. On the other hand, in e-commerce topic, results are slightly better than gaming. Their repository sizes are pretty small. But they have a better class ratio than gaming topics. So, test usage is expected to have a positive relationship with the class counts in the source code.
- How does the test usage change with the size of the repositories in terms of the test methods and overall methods? Answer: Method counts, and test method usage depend on the software projects' characteristics. It stays the same regardless of the topic. However, in method counts behavior, classes can have multiple functionalities or be small-abstract based. The more methods a class has, the more complicated they can become. It is the same for both source code and testing methods. For our two topics, methods are like class counts in the gaming section, and test usage is the same as expected. In the e-commerce section, classes have fewer methods than gaming topics, and test classes have fewer test methods than gaming topics.

The results of this research cannot be generalized neither to the domains under consideration nor to other domains due to the following reasons:

- the study uses a small sample size, and the results are not statistically significant enough to represent the larger population.
- the study sample might not be representative of the whole population.
- the study uses a non-random sampling method, which can introduce bias.
- the study uses a tool, developed by the authors, that might have some inaccuracies or limitations.

V. CONCLUSION

In this research, software projects from GitHub that are written in Java language have been gathered. Then, they were analyzed by code metrics such as source class/methods and (test class/methods. These processes are automated and can be reused for other Maven-based directories. It turns out that Java repositories may differ quite a lot. Also, test usages are unique from project to project. Test usage is essential for quality products and systems, as more robust tests mean fewer bugs, errors, and unintended behavior. It seems that community projects are being deployed without implementing these principles. Our database extracted dozens of repositories because they had zero test cases.

On the other hand, working with SonarQube and Maven has its own benefits, but these also come with some problems. SonarQube is usually used for developing a project from the start and monitoring it. With already existing projects, even though it has the requested structure, there might be version and dependency errors with the API and Maven. In conclusion, these processes can be automated using parsers and third-party tools.

For future work, a database of repositories could be expanded by solving these problems mentioned above. For the analysis, more insight might be used for the clustering to gain a deeper understanding of the correlations of the metrics. Moreover, we plan do an in-depth experimentation in order to find correlations of any statistical significance of findings.

REFERENCES

[1] P. Marinescu, P. Hosek, and C. Cadar, “Covrig: a framework for the analysis of code, test, and coverage evolution in real software,” The Proceedings of the 2014 International Symposium on Software Testing and Analysis, San Jose CA USA: ACM, Jul. 2014, pp. 93–104. doi: 10.1145/2610384.2610419.

[2] A. E. Hassan, “The road ahead for Mining Software Repositories,” The Proceedings of the Frontiers of Software Maintenance, Beijing, China: IEEE, Sep. 2008, pp. 48–57. doi: 10.1109/FOSM.2008.4659248.

[3] C. C. Williams and J. K. Hollingsworth, “Automatic mining of source code repositories to improve bug finding techniques,” IEEE Trans. Softw. Eng., vol. 31, no. 6, pp. 466–480, Jun. 2005, doi: 10.1109/TSE.2005.63.

[4] G. Gousios and D. Spinellis, “Mining Software Engineering Data from GitHub,” The Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina: IEEE, May 2017, pp. 501–502. doi: 10.1109/ICSE-C.2017.164.

[5] V. Cosentino, J. Luis, and J. Cabot, “Findings from GitHub: methods, datasets and limitations,” The Proceedings of the 13th International Conference on Mining Software Repositories, Austin Texas: ACM, May 2016, pp. 137–141. doi: 10.1145/2901739.2901776.

[6] O. Dabic, E. Aghajani, and G. Bavota, “Sampling Projects in GitHub for MSR Studies.” arXiv, Mar. 08, 2021. Retrieved: July, 2023 [Online]. Available from: http://arxiv.org/abs/2103.04682

[7] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D.M. German, and D. Damian, “The promises and perils of mining github,” The Proceedings of the 11th working conference on mining software repositories, ACM, May. 2014, pp. 92-101.

[8] K. K. Chaturvedi, V. B. Sing, and P. Singh, “Tools in Mining Software Repositories,” The 13th Proceedings of the International Conference on Computational Science and Its Applications, Jun. 2013, pp. 89–98. doi: 10.1109/ICCSA.2013.22.

[9] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, “Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining,” Empir. Softw. Eng., vol. 16, no. 3, pp. 325–364, Jun. 2011, doi: 10.1007/s10664-010-9143-7.

[10] K. Mierle, K. Laven, S. Roweis, and G. Wilson, “Mining student CVS repositories for performance indicators,” ACM SIGSOFT Softw. Eng. Notes, vol. 30, no. 4, pp. 1–5, May 2005, doi: 10.1145/1082983.1083150.

[11] A. Arcuri and X. Yao, “A novel co-evolutionary approach to automatic software bug fixing,” The Proceedings of the 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), Jun. 2008, pp. 162–168. doi: 10.1109/CEC.2008.4630793.

[12] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, “Mining Software Repositories to Study Co-Evolution of Production & Test Code,” The Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, Apr. 2008, pp. 220–229. doi: 10.1109/ICST.2008.47.

[13] A. G. Yalçın, “Development of co-evolution tracker tool for software with acceptance criteria,” Master Thesis, Izmir Institute of Technology, 2022. Retrieved: July, 2023 [Online]. Available from: https://gcris.iyte.edu.tr/handle/11147/12714.

[14] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad, “Software Testing Techniques: A Literature Review,” The Proceedings of the 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M), Nov. 2016, pp. 177–182. doi: 10.1109/ICT4M.2016.045.

[15] “SonarQube Tutorial Documentation,” Retrieved: July, 2023 [Online]. Available from: https://docs.sonarsource.com/sonarqube/9.7/extension-guide/web-api/.