TREC: A Regression Test Recommender for Java Projects

Sai Kiran Bhrugumalla
North Dakota State University
Fargo, USA
saikiran.bhrugumalla@ndsu.edu

Ajay Kumar Jha
North Dakota State University
Fargo, USA
ajay.jha.1@ndsu.edu

***Abstract*—Regression testing is critical for maintaining software quality. Therefore, developers must maintain efficient and effective regression test suites. However, the task can be tedious and challenging, especially in Continuous Integration (CI) where developers frequently change code. With each code change, developers may need first to identify all relevant tests for the code change and then identify specific tests among the relevant tests that need to be modified. This process can reduce developers' productivity if they do this manually. Developers may also not modify test suites due to time constraints, resulting in inefficient and ineffective test suites. In this paper, we propose a technique and tool, TREC, that identifies relevant tests for code changes in CI and recommends tests to developers for modifications. TREC performs method and test co-evolution and method call analysis to identify and recommend tests for code changes. We evaluate the effectiveness of TREC by recommending tests for 1,699 developers' modified methods in 437 commits from three open-source Java projects. We find TREC's recommended tests include 2,886 (84.39%) of the 3,420 developers' modified tests for the 1,699 methods, indicating TREC's capability to successfully identify and recommend the majority of tests that need to be modified for the methods.**

Demo: https://www.youtube.com/watch?v=WgIveZuPDBU

***Index Terms*—Test-to-code traceability, test recommendation, test maintenance, regression testing**

I. INTRODUCTION

Change is an essential characteristic of software development [1], and managing change is critical to the continuing usefulness of any software [2]. Software systems go through various types of changes [3]. After each change, developers ideally perform regression testing to ensure that software still works as expected [2], [4]–[6]. Regression testing plays a critical role in maintaining software quality by detecting bugs earlier, especially in Continuous Integration (CI) where developers merge code changes and test them frequently [7]–[9]. Therefore, developers must maintain efficient and effective regression test suites [10].

Developers need to modify tests after code changes [11], [12]. To accomplish the task, developers have to first identify all relevant tests for a code change and then identify tests among the relevant tests that need to be modified. Even for adding new tests, developers have to first identify all relevant existing tests to avoid creating redundant tests. Identifying relevant tests and selecting tests for modification manually is tedious and challenging [13]–[15]. The process can reduce developers' productivity [13], especially in large software systems with thousands or millions of tests [8], [16]. More importantly, developers may not modify tests due to time and resource constraints [7], resulting in inefficient and ineffective regression test suites. This can eventually increase regression testing costs [17] and affect software quality. Therefore, it is important to automate the process [18], [19].

Several techniques have been proposed to create test-to-code traceability (TCT) links to identify relevant tests for a piece of code (i.e., methods and classes) [14], [15], [20]–[23]. Although these techniques effectively create TCT links, they do not create/update TCT links incrementally, meaning developers need to run them on the entire codebase after each change, requiring significant time and resources [14]. Therefore, the techniques are not ideal in CI settings. Moreover, they do not recommend tests for a code change except TestNForce, which is a Visual Studio plugin that shows all the relevant tests for a code change to developers [14]. However, a piece of code may have many relevant tests, and all of them may not be equally important for a code change. Ranking relevant tests based on their importance for a code change can further help developers select tests for modification.

To address the above gaps, we propose a technique and tool, TREC, that identifies TCT links from commit histories and recommends tests for code changes at the method level in CI environments. We hypothesize that the past co-evolution between methods and tests and their co-evolution frequency indicate the relevancy and importance of the tests for future changes in the methods. TREC uses a combination of co-evolution and method-call analysis techniques [24] to extract TCT links from commit histories. TREC ranks TCT links (i.e., tests in the links) for a method based on their frequency in commit histories and recommends the top 5 tests for any changes in a method in a new commit. TREC also identifies any TCT links in a new commit, creating/updating TCT links incrementally, which is the recommended [25] and most desirable feature for a traceability technique [14].

We implement TREC as a command line tool and evaluate the effectiveness of TREC by recommending tests for 1,699 developers' modified methods in 437 commits from three open-source Java projects. The modified methods have 3,420 corresponding developers' modified tests in the commits. We find TREC's recommended top 5 tests for the methods include 84.39% of the developers' modified tests, among which 59.15% and 19.27% are in the first and second ranks, respectively. This shows that the developers could have used the TREC's recommended tests for identifying the majority of tests that needed to be modified for the methods. Our

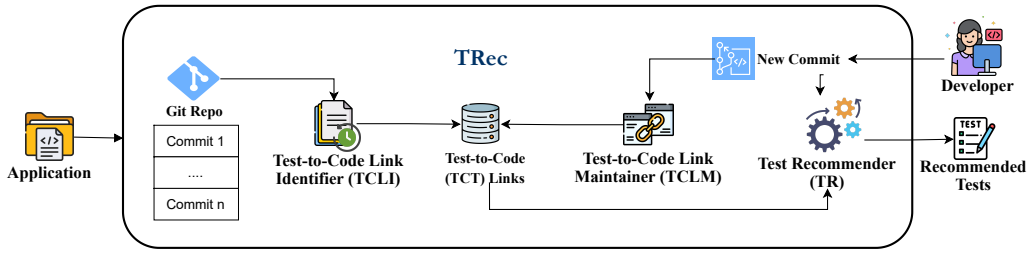

Fig. 1: A high-level overview of TREC

online artifact page contains all the data and tools used in this study [26].

II. APPROACH

Figure 1 shows a high-level overview of TREC. It has three different components: *Test-to-Code Link Identifier* (TCLI), *Test-to-Code Link Maintainer* (TCLM), and *Test Recommender* (TR). TCLI takes a project repository as input, analyzes existing commits in the repository, and extracts TCT links. TCLM maintains the TCT links extracted by TCLI in CI development environments. TCLM takes a new commit as input, identifies any TCT links in this commit, and updates the TCT links. TR also takes the new commit passed to TCLM as input, identifies modified methods that do not have a corresponding added or modified test in this commit, and recommends tests from the TCT links to developers for the modified methods. We now describe each component.

A. TCLI

TCLI analyzes existing commits in projects to identify and extract TCT links at the method level. TCLI first identifies candidate commits in a project that may potentially have TCT links and then analyzes code changes in the candidate commits to extract TCT links. TCLI then organizes and ranks the extracted TCT links for each method based on the frequency of the links in existing commits. TREC runs TCLI only once to collect the historical information.

1) Identify Candidate Commits.: TCLI identifies a commit as a candidate commit if the commit has at least one modified production file and a corresponding modified test file. Developers commonly use the same test filename as the production filename with a *Test* prefix or suffix to create a test file [15]. TCLI leverages this common practice to identify test files for a production file. Specifically, TCLI maps a test file to a production file in a commit if the test filename contains the production filename. TCLI does not perform exact match (e.g., Utilities → UtilitiesTest), because developers may create multiple test files for a production file with some variations in test filenames (e.g., Utilities → UtilitiesTest, UtilitiesBugFixTest).

2) Extract TCT Links.: For each method modified or added in a candidate commit, TCLI finds the corresponding tests modified or added in the candidate commit, establishing TCT links. TCLI leverages JavaParser[1] for this task. Specifically, TCLI compares each modified file in a candidate commit with the previous version of the modified file in the direct parent commit to identify methods or tests added or modified in the candidate commit. TCLI considers a method or test as newly added if it appears in the modified file but not in the previous version of the file. Whereas, TCLI considers a method or test as modified if the body of the method or test (i.e., body converted to string) is not the same in the modified and the previous versions of the file. TCLI considers an added or modified test linked to an added or modified method in the candidate commit if the method is invoked in the test, creating a TCT link.

TCLI also records and maintains the frequency of each unique TCT link in commit histories. Suppose developers modified *foo* method in five different commits, four of these commits also have *testFoo* test modified and the remaining one has *testFooFix* test modified. If TCLI identifies the modified tests are linked to the *foo* method, TCLI extracts two different TCT links with their frequency: {*foo-testFoo, 4*} and {*foo-testFooFix, 1*}.

3) Rank TCT Links: A method may have several corresponding tests in a project, testing different behaviors of the method. However, all the tests may not be equally important for a code change in the method. Therefore, TCLI ranks the extracted TCT links for each method based on the frequency of the links. We assume developers modify some method behaviors and their corresponding tests more often than others. We acknowledge that the approach is not precise. However, an accurate approach would require fine-grained analysis of code changes (e.g., statement and branch), requiring significant time overhead, which is not ideal for CI settings. In the previous example, TCLI ranks *testFoo* first and then *testFooFix* for the *foo* method.

At the end of this process, TCLI produces a database of ranked TCT links for each method in the links.

B. TCLM

Existing techniques that extract TCT links do not incrementally create/update TCT links as software evolves. It means the extracted TCT links may become obsolete and new TCT links will not be identified unless the techniques run from scratch again [25]. Running them from scratch requires significant redundant effort [14]. This may not be even feasible in CI environments, where the code churn rate can be high for large-scale systems [8], [16]. Unlike the existing techniques, TREC uses TCLM to create/update TCT links incrementally in CI environments.

[1]https://javaparser.org/

TCLM takes a new commit as input and extracts TCT links from this new commit as described in Section II-A2. If TCLM finds new TCT links that are not in the TCT link database, TCLM adds the links to the database along with frequency 1. If the links are already in the database, TCLM increments their frequency by 1. This process ensures the identification of any new TCT links and the ranking update to the existing links in CI environments.

C. TR

TR also takes the new commit passed to TCLM as input, identifies candidate methods in the commit for test recommendation, and recommends tests for the candidate methods to developers.

TR considers modified methods in a new commit as candidate methods for test recommendation if they do not have corresponding added or modified tests in the commit. TR checks modified method invocations in all the added/modified tests to identify their corresponding tests. For each identified candidate method, TR then finds all the associated TCT links in the TCT link database and selects the top 5 links based on their frequency. TR then recommends the tests in the top 5 links to developers in the ranked order. A candidate method may have many associated TCT links, resulting in many recommended tests if TR recommends all of them. In this case, developers may have to put significant effort into reviewing the recommended tests, especially if developers are not familiar with the codebase. Therefore, TR recommends only the top 5 tests.

III. Evaluation

To evaluate the effectiveness of TREC in recommending tests for code changes, we conduct a preliminary investigation and answer the following research question:

- **RQ1:** *Does* TREC *recommend the same tests that developers modify for code changes?* We investigate whether TREC's recommended tests are the same as developers' modified tests for code changes. This demonstrates TREC's capability to identify tests that need to be modified for code changes.

For this preliminary investigation, we select three open-source Java projects: commons-lang[2], gson[3], and commons-io[4]. They are popular and actively maintained projects used in previous test evolution studies [12], [27]. Commons-lang and commons-io are also used in a previous traceability study [21]. Table I shows the number of commits and tests in these projects.

A. Method

We run TREC on the projects to extract TCT links from commit histories. We run TREC on a system containing Windows 11 operating system, 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz processor, and 16 GB RAM. TREC creates a database of TCT links for each project. Note that

[2]https://github.com/apache/commons-lang

[3]https://github.com/google/gson

[4]https://github.com/apache/commons-io

TABLE I: Studied projects

Project	Num. of commits	Num. of tests
commons-lang	7,668	4,226
gson	1,986	1,395
commons-io	4,859	2,048
Total	14,513	7,669

TABLE II: Evaluation dataset: developers' modified methods and tests

Project	#Commits	#Modified methods (unique methods)	#Modified tests (unique tests)
commons-lang	256	939 (755)	1,565 (925)
gson	57	134 (96)	518 (281)
commons-io	124	626 (479)	1,337 (639)
Total	437	1,699 (1,330)	3,420 (1,845)

TREC incrementally extracts TCT links from each commit in a project in the temporal order (i.e., old to recent commit), creating/updating TCT links in the database.

To recommend tests, TREC requires a new commit with methods containing code changes. We also need developers' modified tests for the code changes to answer the RQ. Therefore, we create an evaluation dataset that satisfies the above requirements by analyzing existing commits in the studied projects. We first identify those commits where developers modify a method and its tests. The idea is to recommend tests for a modified method and check whether TREC recommends the same tests as modified by developers. We use the same approach described in Section II-A2 to identify commits for creating the evaluation dataset. However, we do not consider added methods/tests for this purpose, because they do not have any modification history; therefore, TREC cannot recommend tests for them. We identify 437 commits in the projects where developers modify 1,699 methods and their 3,420 tests as shown in Table II. The modified methods and tests may not be unique across the commits in a project. We next run TREC on each identified commit to recommend the top 5 tests for each modified method. For the recommendation, TREC considers only those TCT links extracted from the previous commits to the commit for which TREC recommends tests. For example, to recommend tests for commit C_n, TREC considers TCT links extracted from C_1 to C_{n-1} commits. Therefore, C_n is effectively a new commit for TREC.

After the recommendation, we match the top 5 recommended tests with the developers' modified tests for each method. We then calculate TREC's *test recall*, the fraction of developers' modified tests included in the top 5 TREC's recommended tests for all the methods in a project. We also identify the rank at which developers' modified tests appear in the recommended tests. We do not calculate precision because it depends on how many tests developers modify for a method. For example, if developers modify one test for a method, at least four of the five TREC's recommended tests will be incorrect, even though the tests are testing the method.

B. Results

Table III shows the number of TCT links extracted by TREC from the projects. TREC extracts 4,929 unique TCT links for 2,585 methods from 1,086 commits. TREC takes 3 hours 22 minutes, 37 minutes, and 1 hour 48 minutes to extract the TCT links from commons-lang, gson, and commons-io, respectively. Note that this is a one-time effort. Figure 2 shows the frequency of the TCT links in the commits. The median frequency value is one for all the projects. However, some TCT links have high frequency, reaching up to 11 and 5 in commons-lang and commons-io, respectively.

Table IV shows TREC's test recommendation results. We find that TREC's recommended top 5 tests include 2,886 of the 3,420 developers' modified tests across the projects with a test recall of 84.39%. TREC's test recall ranges from 65.64% in gson to 90.29% in commons-lang. Among the 2,886 TREC's recommended tests that match with the developers' modified tests, 1,707 tests (59.15%) are ranked first and 556 (19.27%) are ranked second by TREC in the top 5 recommended lists. TREC's recommended top 5 tests do not include 534 of the 3,420 developers' modified tests. We find two key reasons for this. First, developers modify more than five tests for a method. For example, in commit `5292526` in commons-lang, the developer modifies eight tests for `newThread` method in `BasicThreadFactoryTest` class. However, TREC recommends only five tests for the method, missing three developer's modified tests. Second, some or all of the TREC's recommended top 5 tests for a method are not the same as developers' modified tests. For example, in commit `839b0c2` in gson, the developer modifies four tests for `shouldSkipField` method. in `ExposeAnnotationDeserializationExclusionStrategyTest` class. However, none of the TREC's recommended top 5 tests for this method is the same as the developer's modified tests. Note that TREC successfully extracts TCT links containing all the developers' modified tests for both examples although TREC does not recommend them in the top 5 lists.

***RQ1**:* TREC's recommended tests for the 1,699 modified methods include 84.39% of the tests that developers' modified for the methods. Among these, 59.15% and 19.27% are ranked first and second in the top 5 lists, respectively. This indicates TREC's capability of successfully recommending the majority of tests that needed to be modified for the methods.

C. Threats to Validity

1) Construct Validity: To extract TCT links (Section II-A2), TREC links a test to a method if the method is invoked in the test. A method can be used as a helper method in a test. Therefore, the extracted TCT links in Table III may include helper method and test pairs. However, unlike existing techniques that focus on identifying TCT links for focal methods [15], [21], our goal in this work is to identify TCT links containing method and test pairs that may potentially get modified together, regardless of whether the method is a helper or focal.

TABLE III: TCT links extracted by TREC

Project	#Commits with links	#Methods with links	#TCT Links
commons-lang	654	1,652	2,588
gson	122	177	662
commons-io	310	756	1,679
Total	1,086	2,585	4,929

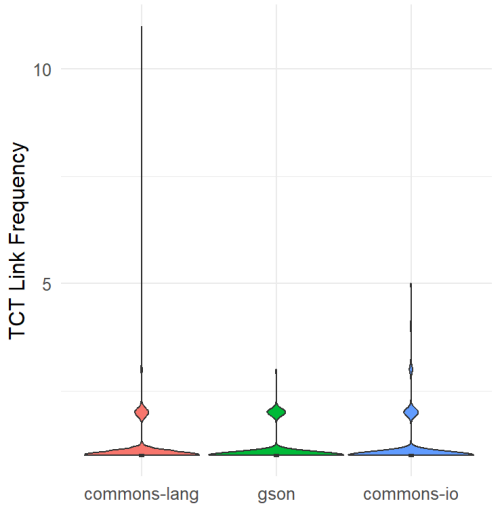

Fig. 2: Frequency of the extracted TCT links

2) Internal Validity: To identify candidate commits (Section II-A1), TREC maps a test file to a production file if the test filename contains the production filename. Although developers commonly write a test filename by prefixing or suffixing the corresponding production filename with Test [15], developers may not follow this naming convention. This may have resulted in TREC not identifying some candidate commits and eventually not extracting TCT links from the commits. However, this does not affect the recommendation results in Table IV, because the evaluation dataset in Table II and the extracted TCT links in Table III do not include methods and tests from potentially missing candidate commits.

3) External Validity: We evaluated TREC by recommending tests for 1,867 methods from three open-source Java projects. The results may not generalize beyond the studied projects. However, this is the first attempt to incrementally extract and update TCT links and recommend tests for modified methods in CI settings.

IV. Related Work

Several studies have proposed different techniques to extract TCT links [24]. Rompaey and Demeyer proposed six different techniques to extract TCT links at the class level [15]. TREC also uses three of them to extract TCT links, but TREC extracts TCT links at the method level. White et al. proposed an approach that combines different dynamic and static analysis techniques to extract TCT links at both method and class levels [21]. They do not use the co-evolution

TABLE IV: Number of developers' modified tests in Table II included in the TREC's recommended top 5 tests

	Rank	commons-lang	gson	commons-io	All
Num. of	1	952	134	621	1,707
matching	2	246	83	227	556
tests in	3	104	55	134	293
different	4	70	40	89	199
ranks	5	41	28	62	131
Total top-5		1,413	340	1,133	2,886
Test recall		90.29	65.64	84.74	84.39

technique, which TREC is mainly based on due to its design to work incrementally in CI environments. Sohn and Papadakis proposed a technique that identifies evolutionary coupling between methods and tests by computing the average time interval between their past changes [23]. The technique then uses the coupling degree to extract TCT links. Unlike this approach, TREC uses method-call analysis to link a test to a method. None of the approaches create/update TCT links incrementally and recommend tests for code changes. Only TestNForce recommends tests for code changes based on TCT links generated through test execution [14]. Unlike TestNForce, TREC not only creates/updates TCT links incrementally but also ranks all the relevant tests for code changes and recommends the top 5 tests for each modified method.

V. Conclusion

In this paper, we proposed TREC, a regression test recommender that identifies tests for methods modified in a commit, ranks the tests identified for each modified method, and recommends the top 5 tests for each modified method to developers. We evaluated TREC by recommending tests for 1,699 developers' modified methods from three open-source Java projects. TREC's recommended tests for the methods included 84.39% of the tests that developers modified for the methods, indicating TREC's capability of successfully identifying and recommending the majority of tests needed to be modified for the methods.

References

[1] M. W. Godfrey and D. M. German, "The past, present, and future of software evolution," in *2008 Frontiers of Software Maintenance*. IEEE, 2008, pp. 129–138.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London, "Incremental regression testing," in *1993 Conference on Software Maintenance*. IEEE, 1993, pp. 348–357.

[3] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of software maintenance and evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[4] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*. IEEE, 1997, pp. 264–274.

[5] H. K. Leung and L. White, "Insights into regression testing (software testing)," in *Proceedings. Conference on Software Maintenance-1989*. IEEE, 1989, pp. 60–69.

[6] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[7] G. Pinto, M. Rebouças, and F. Castor, "Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users," in *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2017, pp. 74–77.

[8] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 233–242.

[9] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

[10] T. Winters, T. Manshreck, and H. Wright, *Software engineering at google: Lessons learned from programming over time*. O'Reilly Media, 2020.

[11] A. Van Deursen and L. Moonen, "The video store revisited–thoughts on refactoring and testing," in *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*. Citeseer, 2002, pp. 71–76.

[12] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, 2012, pp. 1–11.

[13] S. Makady and R. J. Walker, "Validating pragmatic reuse tasks by leveraging existing test suites," *Software: Practice and Experience*, vol. 43, no. 9, pp. 1039–1070, 2013.

[14] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012.

[15] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 209–218.

[16] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagppan, "Fastlane: Test minimization for rapidly deployed large-scale online services," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 408–418.

[17] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 821–830.

[18] D. Ståhl, K. Hallén, and J. Bosch, "Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the eiffel framework."

[19] A. Qusef, "Test-to-code traceability: Why and how?" in *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*. IEEE, 2013, pp. 1–8.

[20] N. Aljawabrah, T. Gergely, S. Misra, and L. Fernandez-Sanz, "Automated recovery and visualization of test-to-code traceability (tct) links: An evaluation," *IEEE Access*, vol. 9, pp. 40 111–40 123, 2021.

[21] R. White, J. Krinke, and R. Tan, "Establishing multilevel test-to-code traceability links," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 861–872.

[22] A. Kicsi, V. Csuvik, and L. Vidács, "Large scale evaluation of natural language processing based test-to-code traceability approaches," *IEEE Access*, vol. 9, pp. 79 089–79 104, 2021.

[23] J. Sohn and M. Papadakis, "Cement: On the use of evolutionary coupling between tests and code units. a case study on fault localization," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 133–144.

[24] R. M. Parizi, S. P. Lee, and M. Dabbagh, "Achievements and challenges in state-of-the-art software traceability between test and code artifacts," *IEEE Transactions on Reliability*, vol. 63, no. 4, pp. 913–926, 2014.

[25] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, and J. Maletic, "The grand challenge of traceability (v1. 0)," *Software and systems traceability*, pp. 343–409, 2012.

[26] TREC Artifacts, https://github.com/STAM-NDSU/TRec, 2024.

[27] T. Virgínio, L. Martins, L. Rocha, R. Santana, A. Cruz, H. Costa, and I. Machado, "Jnose: Java test smell detector," in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, 2020.