Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

12-12-2023

Improving the Test Smell-Based Detection of Flaky Tests

Shubham Kumar Karum
sk4074@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation

Karum, Shubham Kumar, "Improving the Test Smell-Based Detection of Flaky Tests" (2023). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Improving the Test Smell-Based Detection of Flaky Tests

by

Shubham Kumar Karun

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science
in Software Engineering

B. Thomas Golisano College of Computing and
Information Sciences
Rochester Institute of Technology
Rochester, NY

Dec 12, 2023

Approved by:

Dr. Mohamed W. Mkaouer

Dr. Christian Newman

Dr. Ikram Chaabane

Abstract

Regression testing, a critical process in software development, ensures that the recent code changes have not adversely affected existing functionality. A significant challenge in this domain is the existence of flaky tests—tests that inconsistently pass or fail without any changes to the code. These flaky tests undermine the reliability of automated testing and can lead to increased debugging and decreased confidence in software stability. To tackle the issue of flaky tests, there exists approaches that make use of machine learning and prediction models. However, approaches that rely on analyzing test case vocabulary typically face challenges like higher costs to run the tests, a tendency to overfit, and sensitivity to different contexts.

In this research, we conduct an in-depth study on employing test smells and four key flakiness root causes—Async Wait, Concurrency, Test Order Dependency, and Resource Leak—as predictive features for identifying flaky tests. Building upon existing research that validates the use of test smells as flakiness indicators, our study focuses on enhancing the accuracy of the test smell-based model by integrating these four additional flakiness determinants. We rigorously assessed the augmented model's ability to predict flaky tests in a cross-project context and analyzed the information gain contributed by each newly added flakiness root cause. Our findings revealed that particularly the Async Wait and Concurrency categories demonstrated the highest information gain, underscoring their pivotal role in predicting test flakiness. Furthermore, we benchmarked the enhanced test smell-based model against the conventional vocabulary-based approach and the original test smell model, specifically examining improvements in prediction accuracy for flaky tests.

To my beloved parents and my elder brother, This thesis is dedicated to you as a testament to the immeasurable support and faith you have provided me throughout my academic journey. Your unwavering belief in my decisions and the freedom you've granted me to carve my own path have been the cornerstones of my achievements. This accomplishment, while it bears my name, is as much yours as it is mine. It is a reflection of the sacrifices you've made and the unconditional love and support you've never ceased to provide.

With all my love and gratitude,
Shubham Kumar Karun

Contents

1 Introduction **1**

2 Background **4**

2.1 Regression Testing . . . 5
2.2 Flaky Tests . . . 5
2.3 Test Smells . . . 6
2.4 Async Wait . . . 7
2.5 Concurrency . . . 8
2.6 Test Order Dependency . . . 10
2.7 Resource leak . . . 11

3 Research Objective **13**

3.1 Motivation and Contribution . . . 13
3.2 Research Questions . . . 14

4 Related Work **15**

5 Methodology **18**

5.1 Dataset Preparation 18
5.1.1 Data Scraping 19
5.1.2 Pattern Identification 19
5.1.3 Data Merging 20
5.2 Features 20
5.3 Approach 21
5.4 Evaluation Metrics 23

6 Analysis & Discussion 25

7 Threats to Validity 35

8 Conclusion 36

9 Acknowledgement 38

Acknowledgement 38

List of Tables

6.1 Hybrid Feature Based classifier's performance 26
6.2 Test smells-based classifiers' performance 26
6.3 Information gain of each feature of the models 32
6.4 Cross-project Hybrid Test Smell based performance 33
6.5 Cross-project Traditional Test Smell based performance 33
6.6 Cross-project vocabulary-based performance 34

Chapter 1

Introduction

In the field of software engineering, ensuring high-quality software systems is a fundamental objective. Central to achieving this goal is the practice of regression testing, a crucial process extensively utilized by software development teams. This testing method involves a thorough evaluation of system quality by analyzing the test results after implementing code changes. The results of successful tests indicate a functioning system, while failures can signal a reduction in quality. Our research, "Advanced Test Smell-Based Approach for Improved Flaky Test Prediction", focuses on enhancing the methodologies for addressing flaky tests—tests that exhibit inconsistent outcomes—thereby refining the accuracy of regression testing. We present an innovative approach for predicting such flaky tests, evaluating their effectiveness, and providing insights into more reliable and efficient software quality assessment.

The challenge of identifying flaky tests has received significant attention in recent research. To address this challenge, researchers predominantly utilize

two methodologies: Dynamic and Static approaches. The Dynamic approach includes executing test cases multiple times to detect inconsistent outcomes. Despite its effectiveness, this method can be resource-intensive and error-prone, with the added complexity of determining the ideal frequency of test executions. A significant portion of research in this domain leverages machine learning (ML) techniques to assess the probability of test case flakiness, where the choice and analysis of specific ML features critically define each study's uniqueness and success rate.

On the other hand, the Static approach adopts a contrasting strategy. It involves analyzing the test code for flakiness indicators without actual test execution. By analyzing code structure and patterns, this method can efficiently pinpoint potential flakiness, offering a cost-effective and rapid alternative. Nevertheless, its limitation lies in the potential overlooking flakiness aspects that are only observable during live test execution.

Both the Dynamic and Static approaches to flakiness prediction have their respective advantages and drawbacks. The Dynamic approach, though resource-intensive, offers direct observation of flakiness, providing clear insights into test instability. In contrast, the Static approach is more resource-efficient, yet it might overlook flakiness aspects that become apparent only during test execution. The selection between these approaches depends on various factors, including resource availability, time constraints, and specific software requirements.

Pinto et al. [1] introduced an approach for flaky test identification using static features to analyze test code patterns, with the aim of automated detec-

tion. Camara et al. [2] subsequently revisited Pinto et al.'s research, focusing on the real-world application and assessing its performance in cross-project scenarios. Their analysis revealed a key limitation: the vulnerability of the vocabulary-based technique to context sensitivity and overfitting in diverse project environments.

Addressing these shortcomings, Camara et al. [3] proposed an alternative test smell-based method, distinct in its exclusive reliance on static metrics. This model incorporated test case size(LOC), count of test smells, and binary indicators for 19 specific test smells. Their findings indicated that this approach, particularly when using a Random Forest classifier, demonstrated superior performance in cross-project predictions compared to the vocabulary-based model. However, the potential for further improvement was noted, particularly with the addition of more binary features.

Our paper advances Camara et al.'s methodology by integrating four new binary features that represent the primary flaky test root causes: Async Wait, Concurrency, Test Order Dependency, and Resource Leak. An empirical study by Luo et al. [4] suggests that a significant majority of flaky tests can be attributed to these root causes, impacting the Code Under Test (CUT) quality. By expanding the feature set to 25, our study seeks to further refine the predictive accuracy of the test flakiness. Utilizing the same dataset and classifiers as previous studies, this paper evaluates the enhanced model's effectiveness, aiming to set a new benchmark in flaky test prediction.

Chapter 2

Background

This Background section provides an overview of seven fundamental concepts instrumental to our research:

1. Regression Testing
2. Flaky Tests
3. Test Smells
4. Async Wait
5. Concurrency
6. Test Order Dependency
7. Resource Leak

Each concept represents a crucial element in the field of software testing, particularly in identifying and mitigating issues that can compromise software

application integrity. A thorough understanding of these concepts is vital to understand the challenges in software testing and to appreciate the methodologies and solutions proposed in this study.

2.1 Regression Testing

Regression testing ensures that recent code changes have not adversely affected existing features. It involves re-running functional and non-functional tests to verify that the behavior of the software remains consistent after modifications. For example, if a new feature is added to an email application, regression testing would confirm that existing functions, such as sending and receiving emails, are still working as intended.

2.2 Flaky Tests

Flaky tests refer to tests that gives inconsistent results, passing or failing intermittently without any changes to the code. These tests are problematic because they can lead to false positives or negatives, undermining the reliability of the testing process. For instance, a flaky test in a web application might sometimes fail due to network latency issues, even though the application code is correct. We have shared code snippets of flaky tests further in Figure 2.2, Figure 2.3, and Figure 2.4 and Figure 2.4 to describe the 4 primary root causes of Flaky tests i.e Async wait, Concurrency, Test Order Dependency, and Resource Leak as proposed by Luo et al. [4].

2.3 Test Smells

Test smells are patterns in test code that suggest a potential issue, often indicating poor design or maintainability problems. They are analogous to code smells in production code.

Deursen et al. [5] initially identified several test smells, including Assertion Roulette, Eager Test, General Fixture, and Lazy Test, which highlighted prevalent issues in test programming. Building upon Deursen's foundational work, Peruma et al. [6] further extended the scope of test smells. They incorporated additional categories inspired by prevalent shortcomings in unit test programming techniques, as documented in the existing literature. In subsequent research, Peruma and colleagues developed 'TSDetect', an open-source tool specifically designed for the detection of code smells. This tool, TSDetect, has been employed in the current study to identify test smells.

```
@Test
public void testSpinner() {
    for (Map.Entry entry : sourcesMap.entrySet()) {

        String id = entry.getKey();
        Object resultObject = resultsMap.get(id);
        if (resultObject instanceof EventsModel) {
            EventsModel result = (EventsModel) resultObject;
            if (result.testSpinner.runTest) {
                System.out.println("Testing " + id + " (testSpinner)");
                //System.out.println(result);
                AnswerObject answer = new AnswerObject(entry.getValue(), "", new CookieManager(), "");
                EventsScraper scraper = new EventsScraper(RuntimeEnvironment.application, answer);
                SpinnerAdapter spinnerAdapter = scraper.spinnerAdapter();
                assertEquals(spinnerAdapter.getCount(), result.testSpinner.data.size());
                for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                    assertEquals(spinnerAdapter.getItem(i), result.testSpinner.data.get(i));
                }
            }
         }
    }
}
```

Figure 2.1: Test smell example

The code snippet in Figure 2.1 shows a method from the test class EventsScraperTest.java, from the open source TuCanMobile project. This example is a flaky test extracted from the test smells examples mentioned on the testsmell.org website maintained by Peruma et al.

The test method, testSpinner(), contains multiple control statements (i.e. control flow statements). The success or failure of the test is based on the result of the assertion method which is within the control flow blocks and hence not predictable. This also increases the complexity of the test method and hence has a negative impact on maintenance of the test.

2.4 Async Wait

In the context of asynchronous programming, 'Async Wait' is a critical concept where the execution of a program is temporarily halted until a specified condition is fulfilled or an operation completes. In our context, it refers to a category of tests where the tests on execution make an async call and do not adequately wait for the result of the call to become available before using it. Such type of test codes contributes significantly to flaky tests. For example, a test might fail intermittently if it does not correctly wait for an asynchronous API call to complete before asserting the outcome.

The code snippet in Figure 2.2 represents a Async Wait flaky test. This snippet is from the HBase project, used by Luo et al. [4] to demonstrate Async wait root cause in tests (or the CUT). The test relies on a fixed-duration Thread.sleep(2000) to wait for an asynchronous operation to complete before proceeding with assertions. The test assumes that two seconds is always suffi-

```
1 @Test
2 public void testRsReportsWrongServerName() throws Exception {
3   MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
4   MiniHBaseClusterRegionServer firstServer =
5     (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
6   HServerInfo hsi = firstServer.getServerInfo();
7   firstServer.setHServerInfo(...);
8
9   // Sleep while the region server pings back
10  Thread.sleep(2000);
11  assertTrue(firstServer.isOnline());
12  assertEquals(2,cluster.getLiveRegionServerThreads().size());
13  ... // similarly for secondServer
14 }
```

Figure 2.2: Async Wait flaky test

cient time for the firstServer to initialize and respond ("ping back"). However, if the server takes longer than two seconds due to variability in network latency, load, or other environmental conditions, the assertions that follow the sleep will execute before the server is ready, likely causing the test to fail.

This kind of flakiness arises because the test's success is contingent upon an asynchronous event—the server's response—occurring within a predetermined time window, which cannot be guaranteed under all execution circumstances. Thus, it falls into the async wait category of flaky tests, as the timing of the response and the test's execution are not synchronized, leading to non-deterministic test outcomes.

2.5 Concurrency

Concurrency in software engineering is about components or processes executing independently in parallel, which can lead to complex states and behaviors in

an application. For testing, concurrency issues might arise when multiple processes interact in unpredictable ways, causing flaky tests. Common problems include race conditions, where the test result depends on the order of parallel operations, other prevalent concurrency-related problems include atomicity violations, where intended atomic sequences of operations are interrupted by concurrent activities, and deadlocks, which occur when two or more processes are waiting indefinitely for each other to release resources. Tests with such root cause of flakiness fall under the concurrency category.

We next describe a code snippet of a Concurrency flaky test. This snippet

```
1 if (conf != newConf) {
2   for (Map.Entry<String, String> entry : conf) {
3     if ((entry.getKey().matches("hcat.*")) &&
4         (newConf.get(entry.getKey()) == null)) {
5       newConf.set(entry.getKey(), entry.getValue());
6     }
7   }
8   conf = newConf;
9 }
```

Figure 2.3: Concurrency flaky test

The code snippet in Figure Figure 2.3 is from the Hive project, used by Luo et al. [4] to demonstrate Concurrency root cause in the CUT. The code traverses a map that is accessed by multiple threads. Flaky test failures occur when these threads modify the map at the same time, resulting in a ConcurrentModificationException [2].

2.6 Test Order Dependency

Test order dependency occurs when the outcome of a test depends on the sequence in which tests are run, leading to inconsistent results. This typically happens when tests share state between them. For instance, if a test for a user login feature always passes when run after a specific user creation test due to shared state, but fails when run independently, it exhibits a test order dependency.

```
1 @BeforeClass
2 public static void beforeClass() throws Exception {
3   bench = new TestDFSIO();
4   ...
5   cluster = new MiniDFSCluster.Builder(...).build()
6   FileSystem fs = cluster.getFileSystem();
7   bench.createControlFile(fs, ...);
8
9   /* Check write here, as it is required for other tests */
10  testWrite();
11 }
```

Figure 2.4: Test Order Dependency induced flaky test

The code snippet in Figure 2.4 is from the Hadoop project, used by Luo et al. [4] to demonstrate Test Order Dependency root cause in the CUT. The testWrite() method writes data to a file using fs, setting up the data for other tests to read. Initially, the developers assumed that testWrite() would always execute first. However, JUnit does not ensure a specific order of test execution. Consequently, if JUnit runs any of the read tests before testWrite, the test fails due to the absence of the expected data. To resolve this, the developers modified their approach: they removed the standalone testWrite() test and incorporated a call to testWrite() within the beforClass() method, as indicated

in line 10. This change ensures that testWrite() executes once before any other tests in the class, preparing the necessary data setup for subsequent tests.

2.7 Resource leak

Resource leaks occur when an application fails to properly manage system resources, such as file handles or memory allocations. These leaks can cause flaky tests that pass or fail unpredictably, depending on the availability and state of the resource at the time of test execution. The next figure presents an illustrative example of a resource leak within a test case designed to assess resource allocation.

```
1   DatabaseProvider ssp;
2   String currentDirectory = ...;
3   [TestMethod]
4   public void ResourceAllocation() {
5     ssp = new DatabaseProvider(currentDirectory);
6     ...
7   }
8   [TestCleanup]
9   public void TestCleanup() {
10    ClearConnections();
11    if (File.Exists(dbPath)) {
12      File.Delete(dbPath);
13    }
14    ...
15  }
```

Figure 2.5: Resource Leak induced flaky test

The code snippet in the example Figure 2.5 is used by Lam et al. [7], to demonstrate Resource leak root cause in the CUT. ResourceAllocation() tests

whether the necessary resources are properly allocated for an application. The application internally uses a third-party database to store some information. To ensure isolation between test cases, the TestCleanup() method is designated to delete the database file, allowing for a fresh environment for each subsequent test. However, an issue arises due to the behavior of the third-party database library used by the application. This library does not immediately release the file handle upon request; it requires the garbage collector to execute before the file handle is released. Consequently, if the garbage collector has not run by the time File.Delete(dbPath) is invoked on Line 12, an exception is thrown because the file is still in use, leading to a flaky test outcome.

Chapter 3

Research Objective

3.1 Motivation and Contribution

Our research builds on the work of Camara et al. [3], which showed that using test smells can help predict when tests might be flaky. Their approach showed promise in cross-project validations, achieving nearly the same results (55%) as state-of-the-art vocabulary-based methods (56%). However, test smell-based models were up to 14% less precise than vocabulary-based models in some cases. Despite this, they performed notably well in the intra-project context, with 17% more accuracy in the best case.

Our study aims to enhance the traditional test smell-based prediction model by integrating new static features to improve accuracy. We strive to significantly advance our approach's performance in the inter-project context and create a more dependable and generalized approach for identifying potentially flaky tests.

3.2 Research Questions

- **RQ1:** ***What is the predictive accuracy of the Hybrid Test Smell-Based Model for test flakiness compared to the Traditional Test-Smell-Only Approach?***

  This question seeks to evaluate the efficacy of the Hybrid Model, which incorporates both test smells and additional features, against the conventional approach that relies solely on test smells for predicting test flakiness.

- **RQ2:** ***To what extent are the newly integrated features correlated with the prediction of test flakiness?***

  The focus here is to quantify the association between the newly added features and their impact on the accuracy of test flakiness predictions, thereby determining their predictive strength and significance.

- **RQ3:** ***How do the results of the Hybrid Test Smell-Based Model compare with the existing Vocabulary-Based Model and the Traditional Test Smell-Based Model in a cross-project validation context?***

  This question aims to benchmark the performance of the Hybrid Model in predicting flakiness across different projects, contrasting it with the results obtained from the Vocabulary-Based Model and the Traditional Test Smell-Based approach to understand its relative performance in a broader context.

Chapter 4

Related Work

Many studies addressed challenges to software maintenance in general [8–15, 15–90, 90, 91, 91, 92, 92, 93, 93–117], and test flakiness in particular. The issue of flaky tests has emerged as a significant concern in software engineering, primarily due to its adverse impact on developer productivity and the overall software development lifecycle. Flaky tests, characterized by their non-deterministic nature under the same conditions, undermine the reliability of test suites and lead to increased maintenance efforts. Developers often find themselves repeatedly re-running tests to distinguish between genuine failures and flakiness, a process that is not only time-consuming but also diverts attention from critical development tasks. This constant need for verification and debugging can significantly slow down the development process, leading to delayed releases and increased costs. Furthermore, the presence of flaky tests can erode trust in the testing process itself, making it difficult for teams to rely on automated tests for continuous integration and deployment. These challenges explain the

sustained interest and ongoing efforts within the research community to address flaky tests [118] [119] [120]. By developing more effective detection and mitigation strategies, researchers aim to enhance the efficiency and accuracy of testing processes, ultimately contributing to more stable software products and more productive development environments.

W. Lam, R. Oei, and A. Shi [121] [122] have proposed adopting dynamic approaches which involve running the test suite for a fixed number of times. However, it increases the cost of execution and for large organizations, this becomes a scaling issue.

According to Pinto et al. [1], it is possible to extract a vocabulary of patterns of words from the test code that can be used to determine whether a test is flaky or not. The authors developed a dataset of Java projects with test cases labeled as flaky and non-flaky and then used it to train and evaluate ML systems. Overall, all classifiers performed well, with SVM having the best recall and Random Forest having the best precision (0.99) and F1-Score (0.92). The top 20 features with the best information gain are also displayed in the study. The effectiveness of the vocabulary-based strategy in Python projects was also the subject of several research [123] [124].

Camera et al. [2] replicated the Pinto et al. [1] study. They extended the work by using the trained classifiers to predict flaky tests using a different test dataset. The classifiers were used for prediction in two different contexts namely Inter and Intra project contexts. Among the trained classifiers, LDA achieved the best results with recall of 0.75 in intra-project contexts and 0.45 in inter-project scenarios. The authors concluded that the vocabulary-based

approach, while effective in some settings, is sensitive to context and has a tendency to overfit.

Alshammari et al. [125] developed an approach, FlakeFlagger, to predict test flakiness, incorporating both static and dynamic features like test smells, test coverage, and source code management. They used a dataset of 24 open-source Java projects to benchmark FlakeFlagger against both a vocabulary-based [1] and a combination of both. While recall rates were comparable across all three approaches (74%, 72%, and 74%), FlakeFlagger displayed a significant 49% improvement in precision over the vocabulary-based approach. The hybrid approach further improved precision by 6%. Interestingly, although FlakeFlagger includes its own expanded test smell detector, the analysis revealed a low correlation between these test smells and test flakiness.

Camera et al. [3] evaluated the performance of the traditional test smell-based approach for prediction of flakiness. It was observed that using test smells are potentially good predictors of flakiness. The authors also compared their results with the vocabulary-based approach. The test smell-based models had the precision(83%) which is 14% lower than the state-of-the-art vocabulary based approach(97%). In the cross-project validation, the test smell-based approach in general performed better in the intra- and inter- project contexts.

The test smell-based approach produced promising results and opened the possibility to further add more static or dynamic features to the test smells as indicators. We extended Camara et al. [3]study on using test smells and explored the possibility of extending the features for predicting flaky tests.

Chapter 5

Methodology

5.1 Dataset Preparation

In our methodology, we begin with the foundational dataset utilized by Camara et al. [3], comprising 2932 flaky and 1400 non-flaky tests. Specifically, we included 1377 flaky and 1400 nonflaky tests from the'msr4flakiness' dataset, while incorporating an additional 155 flaky tests from the 'idFlakies' dataset, both of which were previously assembled by Pinto et al. To gather the test case class files associated with all test methods from Camara et al.'s dataset, we cloned the GitHub repository "https://github.com/ncsu-swat/msr4flakiness". This repository organized dataset into three distinct sections: i) flaky test case class files from 'msr4flakiness', ii) non-flaky test case class files from 'msr4flakiness', and iii) flaky test case class files from 'idFlakies'.

Following the initial data collection, we followed the planned steps designed to refine and augment the dataset for our analysis. These steps are outlined

in the following subsections of our methodology.

5.1.1 Data Scraping

Our first task involved the extraction of test methods, classified as either flaky or non-flaky, from the collected test case class files. The code snippets for each test method were scraped and stored separately, aligning with the categorization into the three sections: flaky test cases from 'msr4flakiness', non-flaky test cases from 'msr4flakiness', and flaky test cases from 'idFlakies'.

5.1.2 Pattern Identification

Informed by the findings of Luo et al. [4], we acknowledged Async Wait, Concurrency, and Test Order Dependency as the primary root causes of flaky tests. To these established categories, we introduced an additional parameter: Resource Leak. Based on the survey of existing research on flakiness prediction by Parry et al. [126], resource leaks were an important contributing factor to test flakiness in a large-scale study of open-source Java projects. We developed regex patterns to identify test methods associated with these four root causes within the code snippets. This identification process was executed separately for flaky tests in 'msr4Flakiness', non-flaky tests in 'msr4Flakiness', and flaky tests in 'idFlakies', ensuring a comprehensive analysis across varying test scenarios.

5.1.3 Data Merging

The result of our pattern identification phase was the derivation of four new feature columns: “AsyncWait”, “Concurrency”, “TestOrderDependency”, and “ResourceLeak”. These columns were then merged into the base dataset ’Sampled.csv’, creating an improvised dataset that incorporated the newly identified features. This enhanced version of ’Sampled.csv’, now equipped with additional features, forms the base of our research study.

5.2 Features

Existing Features

The dataset we utilized already comprised 21 features. Of these, 19 are specific test smells associated with each test case, capturing a broad spectrum of potential issues within the test code. The remaining two features focus on code complexity: LOC (Lines of Code), which measures the length of the code, and Smell Count, which aggregates the total number of test smells present. These features provide a foundational understanding of the test environment and its inherent complexities. [Reference: Test Smells Table]

New Features

To enhance the predictive power of our model, we have introduced four new binary features, each addressing a critical root cause of the test flakiness:

i) **Async Wait**: Given the prevalence of asynchronous operations in modern software, flaky tests arising from asynchronous waits are a key area of con-

cern. Issues such as race conditions or improper handling of async callbacks are common. Identifying asynchronous waits is vital due to their susceptibility to intermittent failures, which may vary with environmental changes or across different test runs.

ii) **Concurrency**: Concurrency introduces complexities in testing, raising the potential for issues like deadlocks, race conditions, and resource contention. Tests involving concurrent executions are prone to unpredictable behavior, especially in multi-threaded environments. Analyzing these tests sheds light on concurrency-related issues, making it an essential feature for our study.

iii) **Test Order Dependency**: This feature focuses on the dependency of a test's outcome on the sequence of test execution. Identifying tests that are affected by shared states or other tests' side effects is crucial, as these are common sources of unpredictability in test suites.

iv) **Resource Leak**: A Resource Leak is identified as the root cause of a flaky test when the test's outcome is directly influenced by the application's failure to properly handle resources, such as acquiring or releasing file locks.

These new features, representing the main root causes of flakiness in test methods, have been chosen to augment our dataset. The decision is aimed at significantly refining the model's ability to predict flaky tests [126] [4].

5.3 Approach

Our approach in this research closely mirrors the experimental setup used by Camara et al. [3], with some key enhancements. Initially, like Camara et al., we use the dataset from Pinto et al. [1] to construct our predictive models.

For cross-project validation, we incorporate data from Lam et al. [121], which includes flaky tests from 72 different projects.

Camara et al.'s dataset, processed through the tsDetect tool, yielded 19 test smell features plus two code complexity features: LOC (Lines of Code) and Smell Count (total number of test smells). They created two distinct datasets for model training and testing (with 1377 flaky and 1400 non-flaky samples) and for cross-project validation.

Building upon this foundation, we introduce four additional features to the existing dataset. Our enriched dataset now comprises 1377 flaky and 1400 non-flaky samples from "msr4flakiness," and 153 flaky samples from "idFlakies," totaling 25 features. The "msr4flakiness" samples are designated for training the models, while the "idFlakies" samples are reserved for cross-project validation.

In preparation for model training, we first conduct exploratory data analysis using libraries like Matplotlib. This involves data preprocessing steps like converting feature columns to integers and transforming the dataset into a numeric format. This conversion is crucial for the model to effectively interpret the features and the ground truth. We also visualize a correlation matrix to discern relationships between features.

An essential part of our methodology is hyperparameter tuning, performed using GridSearchCV. This process involves systematically experimenting with different combinations of classifier parameters to optimize performance. The goal here is to enhance the model's predictive ability while avoiding pitfalls like overfitting or underfitting.

Our models are then trained iteratively using the training set. We selected a range of classifiers, including Random Forest, for their proven efficacy in classification tasks and their suitability for handling our dataset's specific characteristics. Random Forest, in particular, showed the best performance in our initial trials. The final step involves testing the model for cross-project validation, allowing us to directly compare our results with those obtained by Camara et al. and Pinto et al., thereby validating the effectiveness of our enhanced approach.

5.4 Evaluation Metrics

To assess the effectiveness of our predictive models, we employed a comprehensive suite of evaluation metrics, each offering a unique perspective on the model's performance.

Precision: This metric calculates the ratio of correctly predicted flaky tests to the total predicted flaky tests. High precision indicates a low rate of false positives.

Recall: Also known as sensitivity, recall measures the ratio of correctly predicted flaky tests to the actual flaky tests in the dataset. It reflects the model's ability to detect all relevant instances.

F1-Score: Representing the harmonic mean of precision and recall, the F1-Score provides a balance between these two metrics, particularly useful when the class distribution is imbalanced.

MCC (Matthews Correlation Coefficient): This coefficient offers a comprehensive measure of the model's quality, taking into account true and

false positives and negatives. MCC is particularly insightful in evaluating binary classification problems.

AUC (Area Under the ROC Curve): The AUC represents the degree of separability achieved by the model. It measures the model's ability to distinguish between classes, with higher values indicating better classification performance.

For a more granular analysis, we utilized tools such as confusion matrices and ROC (Receiver Operating Characteristic) curves. These visual representations are instrumental in understanding the trade-offs between true positive and false positive rates, thus providing a more nuanced view of the model's performance.

In our cross-project validation, specifically the intra- and inter-project evaluations, we focused on the 'idFlakies' dataset. Given this dataset's lack of non-flaky test examples, recall was the primary metric used for evaluation. This approach ensures that our assessment aligns with the dataset's structure and provides meaningful insights into the model's capability to correctly identify flaky tests.

Chapter 6

Analysis & Discussion

RQ1: *What is the predictive accuracy of the Hybrid Test Smell-Based Model for test flakiness compared to the Traditional Test-Smell-Only Approach?*

Upon reviewing the results, the Hybrid Test Smell-Based Model [Table 6.1] shows a clear improvement over the Traditional Test-Smell-Only approach [Table 6.2]. For instance, with the Hybrid Model, the Random Forest classifier achieved an accuracy, precision, and recall of approximately 85%, with the MCC at a strong 0.71 and the AUC at 0.93. These figures represent a tangible increase in performance metrics compared to the Traditional Model, where the Random Forest's precision and recall were both at 0.83, and the AUC was at 0.90.

Moreover, the Hybrid Model displayed consistently higher MCC values, with most classifiers achieving scores above 0.60, indicating a strong correlation between predicted and actual values. In comparison, the Traditional Model

Algorithm	Precision	Recall	F1	MCC	AUC
Random Forest	**0.85**	**0.85**	0.85	0.71	0.92
Decision Tree	0.84	0.84	0.84	0.68	0.87
KNN	0.81	0.81	0.81	0.63	0.87
LR	0.81	0.81	0.81	0.62	0.88
Perceptron	0.81	0.81	0.81	0.62	N/A
LDA	0.80	0.80	0.79	0.60	0.87
SVM	0.75	0.75	0.75	0.51	0.84
Naive Bayes	0.75	0.70	0.68	0.45	0.83

Table 6.1: Hybrid Feature Based classifier's performance

Algorithm	Precision	Recall	F1	MCC	AUC
Random Forest	0.83	0.83	0.83	0.65	0.90
Decision Tree	0.83	0.83	0.83	0.66	0.86
KNN	0.81	0.81	0.81	0.62	0.81
LR	0.79	0.79	0.79	0.59	0.87
LDA	0.78	0.78	0.78	0.56	0.86
Perceptron	0.78	0.78	0.78	0.55	0.86
SVM	0.75	0.75	0.75	0.50	0.83
Naive Bayes	0.74	0.65	0.61	0.37	0.78

Table 6.2: Test smells-based classifiers' performance

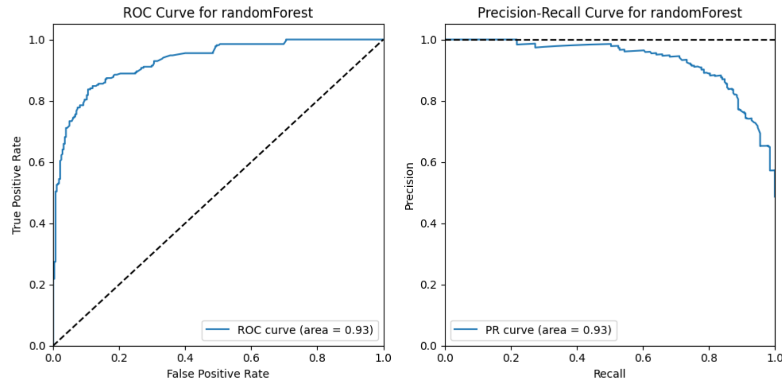

Figure 6.1: Curves for the best performing classifier

had lower MCC values, with the Naive Bayes classifier scoring as low as 0.37. In Figure 6.1 we can see the ROC curve and Precision-Recall curve for the Random Forest classifier in Hybrid approach.

In summary, the data indicates that the Hybrid Test Smell-Based Model outperforms the Traditional Test-Smell-Only Approach. With the addition of new features accounting for root causes of flakiness, the Hybrid Model enhances predictive accuracy, precision, and recall, affirming the value of integrating these additional features into the predictive framework.

RQ2: *To what extent are the newly integrated features correlated with the prediction of test flakiness?*

We've analyzed the information gain of each feature within our hybrid feature-based model. As depicted in figure6, we calculated the information gain based on each feature's entropy, which is a measure of its predictive value regarding test flakiness. The Total Occurrences column in the table quantifies the number of times each feature appears in our dataset, while 'Total Flaky Occurrences' and 'Total Non-flaky Occurrences' columns provide a breakdown of these occurrences across flaky and non-flaky tests, respectively.

Significantly, our new features, specifically 'Async Wait' and 'Concurrency', rank within the top five for information gain among all features in our hybrid model. This shows a strong correlation between the new features and the likelihood of test flakiness, highlighting their importance in improving our model's performance. Furthermore, it's noteworthy that for three of the newly added features—'Concurrency' with 87.57%, 'Async Wait' with 87.76%, and 'Resource Leak' with 92%—a vast majority of the affected tests are classified as flaky. This correlation suggests that these features are potent indicators of flakiness.

In conclusion, our analysis confirms that the newly integrated features are significantly associated with flaky tests. The presence of two of these features among the top five [Fig.6.2] with the highest information gain reinforces their predictive strength and validates their inclusion in our enhanced model.

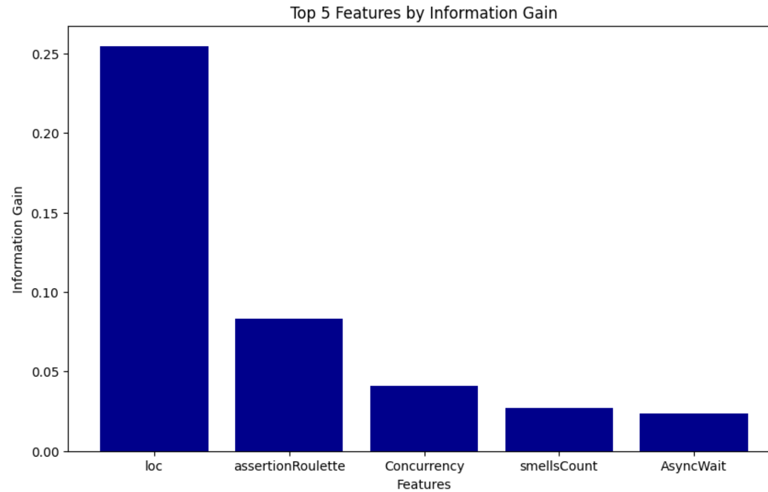

Figure 6.2: Top 5 features by Information gain

RQ3: *How do the results of the Hybrid Test Smell-Based Model compare with the existing Vocabulary-Based Model and the Traditional Test Smell-Based Model in a cross-project validation context?*

We engaged in a thorough examination of the models' performance using the idFlakies dataset for cross-validation. In the intra-project context, the hybrid feature-based model [Table 6.4] shows superior performance with three classifiers reaching recall values above 70%, as opposed to the traditional test smell-based approach [Table 6.5] where only two classifiers exceed that threshold. Notably, both Logistic Regression (LR) and Perceptron are the standout classifiers in the hybrid model, each with a recall of 0.74, which is consistent with the best-performing classifier in the traditional model.

Moving to the inter-project context, the traditional test smell-based model exhibits a slightly stronger performance, with its best classifier, SVM, achieving a recall of 0.55. This contrasts with the hybrid model, where SVM shows a slightly lower recall of 0.51.

When we compare the hybrid model [Table 6.4] to the vocabulary-based approach [Table 6.6] within the intra-project scope, we observe that the hybrid model's LR and Perceptron classifiers outperform with recall values of 74%, against the 57% recall achieved by the vocabulary approach's best classifier, KNN. This indicates that, generally, classifiers from the hybrid feature-based model tend to yield better recall values in the intra-project validation.

In terms of the inter-project validation, the hybrid model's highest recall is 51% with SVM, which is marginally lower than the 56% achieved by the LDA classifier in the vocabulary-based model. However, a broader evaluation across all classifiers suggests the hybrid model generally fares better in cross-project validation than the vocabulary-based approach.

In conclusion, the hybrid feature-based approach demonstrates enhanced performance in the intra-project context compared to both the traditional test smell-based and vocabulary-based approaches. Conversely, in the inter-project context, despite the best classifier from the hybrid model having a slightly lower recall, the overall performance suggests an advantage for the hybrid feature-based approach in cross-project validation scenarios.

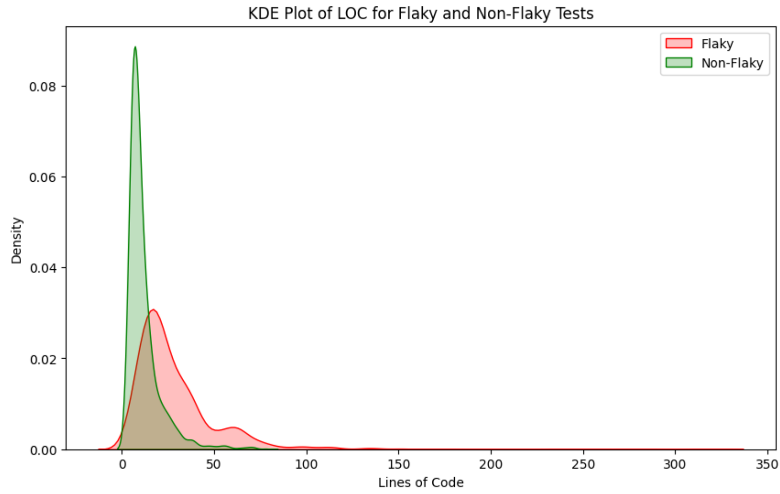

Figure 6.3: KDE Plot of LOC for Flaky and Non-Flaky Tests

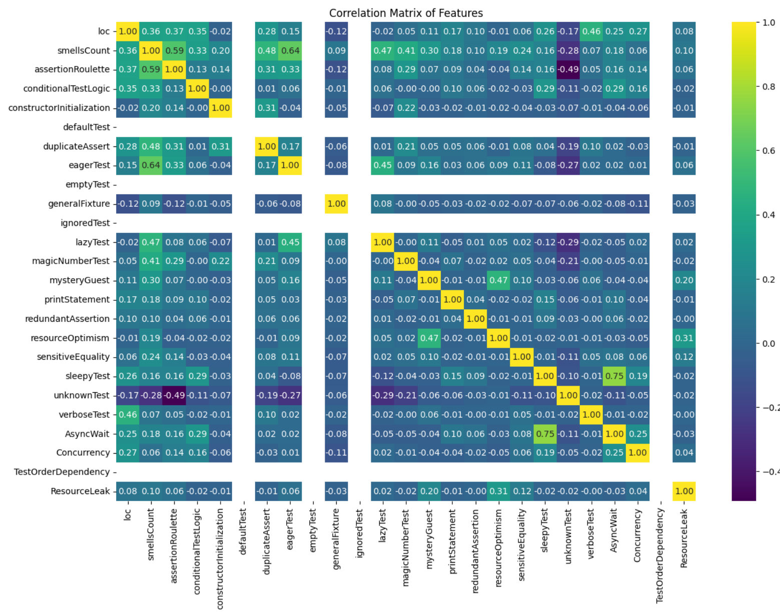

Figure 6.4: Correlation Matrix of features

Pos.	Features	Inf. Gain	Total	Flaky	% Flaky
1	LOC	0.254 817 1	2775	1377	49.58
2	Assertion Roulette	0.832 619	1388	968	69.75
3	Concurrency	0.470 961	314	275	87.58
4	Smells Count	0.270 147	2653	1356	51.11
5	Async Wait	0.236 097	188	165	87.77
6	Sleepy Test	0.194 695	112	105	93.75
7	General Fixture	0.160 650	267	61	22.85
8	Duplicate Assert	0.154 864	376	269	71.54
9	Constructor Initialization	0.103 536	68	63	92.65
10	Print Statement	0.105 576	58	55	94.83
11	Sensitive Equality	0.840 032	129	95	73.64
12	Lazy Test	0.440 699	1786	817	45.74
13	Resource Optimism	0.426 438	75	17	22.67
14	Conditional Test Logic	0.419 574	356	219	61.52
15	Resource Leak	0.320 018	25	23	92.00
16	Unknown Test	0.213 505	544	234	43.01
17	Verbose Test	0.177 088	7	7	100.00
18	Magic Number Test	0.109 571	411	227	55.23
19	Mystery Guest	0.547 207	124	71	57.26
20	Eager Test	0.245 867	970	496	51.13
21	Redundant Assertion	0.827 896	8	4	50.00
22	Ignored Test	0	0	0	0.00
23	Empty Test	0	0	0	0.00
24	Default Test	0	0	0	0.00
25	TestOrderDependency	0	0	0	0.00

Table 6.3: Information gain of each feature of the models

Algorithm	Intra-Project			Inter-Project		
	Recall	TP	FN	Recall	TP	FN
Random Forest	0.71	25	10	0.47	57	63
Decision Tree	0.65	23	12	0.35	42	78
KNN	0.48	17	18	0.40	48	72
LR	0.74	26	9	0.47	57	63
Perceptron	0.74	26	9	0.44	53	67
LDA	0.65	23	12	0.47	57	63
SVM	0.65	23	12	0.51	62	58
Naive Bayes	0.57	20	15	0.20	42	78

Table 6.4: Cross-project Hybrid Test Smell based performance

Algorithm	**Intra-Project**			**Inter-Project**		
	Recall	TP	FN	Recall	TP	FN
Random Forest	0.69	24	11	0.54	65	55
Decision Tree	0.66	23	12	0.54	65	55
KNN	0.51	18	17	0.51	61	59
LR	0.74	26	9	0.48	57	63
Perceptron	0.71	25	10	0.48	57	63
LDA	0.66	23	12	0.47	57	63
SVM	0.66	23	12	0.55	66	54
Naive Bayes	0.57	20	15	0.14	17	103

Table 6.5: Cross-project Traditional Test Smell based performance

Algorithm	**Intra-Project**			**Inter-Project**		
	Recall	TP	FN	Recall	TP	FN
Decision Tree	0.31	11	24	0.39	47	73
LDA	0.29	10	25	0.56	67	53
LR	0.20	7	28	0.30	36	84
Random Forest	0.17	6	29	0.29	35	85
Naive Bayes	0.17	6	29	0.13	15	105
SVM	0.09	3	32	0.17	20	100
KNN	0.57	20	15	0.23	27	93
Perceptron	0.34	12	23	0.33	40	80

Table 6.6: Cross-project vocabulary-based performance

Chapter 7

Threats to Validity

Construct Validity: In some instances, tsDetect [127] might miss determining the production class during the pre-processing of the test code to retrieve the test smells. As a result, the smells would not be extracted, which could compromise the outcome.

Internal Validity: When relating independent and dependent variables, it could skew the findings. The lack of non-flaky classes in the data set between projects can make it impossible to gather precision and other measures to use as a reference.

External validity: As in this study, we are targeting the Java language and a limited set of project domains, we cannot generalize the results. Also, there could be a difference in the performance of models if there is a significant difference in the size of the dataset for Intra-project and cross-project datasets.

Chapter 8

Conclusion

Regression testing plays a critical role in the continuous delivery of high-quality software, and the presence of flaky tests poses significant challenges to both development processes and software quality. Addressing this issue, our study delves into the potential of using extended sets of features as predictors of flakiness in tests along with test smells. Through comprehensive research and analysis, utilizing standard evaluation metrics outlined in this paper, we have assessed the efficacy of using these extended sets of features as predictors for flaky tests. Our experiments identified that training models to identify flaky root causes such as Async wait and Concurrency played a significant role in further predicting the flaky tests. This approach is benchmarked against the test smell and the vocabulary-based approach in the cross project scenario making it a more generalized approach for cross project validation.

Our findings suggest that Async wait and Concurrency are indeed valuable indicators of potential test flakiness, as evidenced by the Information gain

results of the flaky root causes used as predictors of flaky tests. It should be noted especially the 4% increase in the recall rate for inter-project scenarios. Looking ahead, expanding the training dataset with a greater number of projects presents an opportunity to enhance the accuracy of this approach. Moreover, while this study primarily focuses on static methods, future research could benefit from exploring a combination of dynamic approaches with our static approach, potentially offering a more holistic view of test flakiness and its predictors.

Chapter 9

Acknowledgement

I would like to express my deepest gratitude to my faculty advisor, Dr. Mohamed Wiem Mkaouer, for his unwavering support and invaluable guidance throughout the course of this thesis study.

Dr. Mkaouer's expertise and insightful perspectives in test smells have been a beacon of light as I navigated through the depth of my research. His encouragement and belief in my capabilities empowered me to push the boundaries of my knowledge and to aspire for excellence. His patience and availability, despite his many commitments as Associate Professor and Director, made my journey not only possible but also a profound learning experience.

This thesis stands as a testament to Dr. Mkaouer's commitment to nurturing his students and his dedication to the advancement of scholarship. For all this and more, I am eternally thankful!

Shubham Kumar Karun

Bibliography

[1] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 492–502, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Bruno Henrique Pachulski Camara, Marco Aurélio Graciotto Silva, Andre T. Endo, and Silvia Regina Vergilio. What is the vocabulary of flaky tests? an extended replication. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 444–454, 2021.

[3] Bruno Camara, Marco Silva, Andre Endo, and Silvia Vergilio. On the use of test smells for prediction of flaky tests. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing*, SAST '21, page 46–54, New York, NY, USA, 2021. Association for Computing Machinery.

[4] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIG-*

SOFT International Symposium on Foundations of Software Engineering, FSE 2014, page 643–653, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Arie Deursen, Leon M.F. Moonen, A. Bergh, and Gerard Kok. Refactoring test code. Technical report, NLD, 2001.

[6] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[7] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 101–111, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Anthony Peruma, and Stephanie Ludi. Do the test smells assertion roulette and eager test impact students' troubleshooting and debugging capabilities? *arXiv preprint arXiv:2303.04234*, 2023.

[9] Christian D Newman, Michael J Decker, Reem Alsuhaibani, Anthony Peruma, Mohamed Mkaouer, Satyajit Mohapatra, Tejal Vishoi, Marcos Zampieri, Timothy Sheldon, and Emily Hill. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering*, 2021.

[10] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1303–1313, 2021.

[11] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, page e2395, 2021.

[12] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology*, 140:106675, 2021.

[13] Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian Newman, and Mohamed Wiem Mkaouer. Satdbailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022.

[14] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):1–43, 2022.

[15] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. On the documentation of refactoring types. *Automated Software Engineering*, 29(1):1–40, 2022.

[16] Eman Abdullah Alomar, Tianjia Wang, Vaibhavi Raut, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: an empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2022.

[17] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Christian D Newman. Using grammar patterns to interpret test method name evolution. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 335–346. IEEE, 2021.

[18] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Mining and managing big data refactoring for design improvement: Are we there yet? *Knowledge Management in the Development of Data-Intensive Systems*, pages 127–140, 2021.

[19] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[20] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 347–357. IEEE, 2019.

[21] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. Learning to recommend third-party library migration opportunities at the api level. *Applied Soft Computing*, 90:106140, 2020.

[22] Hussein Alrubaye, Stephanie Ludi, and Mohamed Wiem Mkaouer. Comparison of block-based and hybrid-based environments in transferring programming skills to text-based environments. *arXiv preprint arXiv:1906.03060*, 2019.

[23] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian Donald Newman. Contextualizing rename decisions using refactorings and commit messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 74–85. IEEE, 2019.

[24] Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. A study on developer perception of transformation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 34–41, 2018.

[25] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the detection of third-party java library migration at the function level. In *CASCON*, pages 60–71, 2018.

[26] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. Variability in library evolution: An exploratory study on open-source java libraries. In *Software Engineering for Variability Intensive Systems*, pages 295–320. Auerbach Publications, 2019.

[27] Montassar Ben Messaoud, Ilyes Jenhani, Nermine Ben Jemaa, and Mohamed Wiem Mkaouer. A multi-label active learning approach for mobile app user review classification. In *International Conference on Knowledge Science, Engineering and Management*, pages 805–816. Springer, 2019.

[28] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. Migrationminer: An automated detection tool of third-party java library migration at the method level. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*, pages 414–417. IEEE, 2019.

[29] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. Price: Detection of performance regression introducing code

changes using static and dynamic metrics. In *International Symposium on Search Based Software Engineering*, pages 75–88. Springer, 2019.

[30] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[31] Licelot Marmolejos, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, pages 1–17, 2021.

[32] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58. IEEE, 2019.

[33] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Increasing the trust in refactoring through visualization. In *2020 IEEE/ACM 4th International Workshop on Refactoring (IWoR)*, 2020.

[34] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications*, 167:114176, 2021.

[35] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[36] Eman Abdullah AlOmar, Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 342–349, 2020.

[37] Eman Abdullah AlOmar, Philip T Rodriguez, Jordan Bowman, Tianjia Wang, Benjamin Adepoju, Kevin Lopez, Christian Newman, Ali Ouni, and Mohamed Wiem Mkaouer. How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse*, pages 261–276. Springer, 2020.

[38] Eman Abdullah AlOmar, Tianjia Wang, Raut Vaibhavi, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: An

empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2021.

[39] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 350–357, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[42] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages

using multi-label active learning. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1760–1767, 2019.

[43] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[44] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 331–336, 2014.

[45] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1263–1270, 2014.

[46] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, 2016.

[47] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, 22(2):894–927, 2017.

[48] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, and Ali Ouni. Recommending relevant classes for bug reports using multi-objective search. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 286–295. IEEE, 2016.

[49] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 193–202, 2019.

[50] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1):1–61, 2022.

[51] Wajdi Aljedaani, Mona Aljedaani, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Stephanie Ludi, and Yousef Bani Khalaf. I cannot see you—the perspectives of deaf students to online learning during covid-19 pandemic: Saudi arabia case study. *Education Sciences*, 11(11):712, 2021.

[52] Islem Saidani, Ali Ouni, and Wiem Mkaouer. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*, 2021.

[53] Marwa Daaji, Ali Ouni, Mohamed Mohsen Gammoudi, Salah Bouktif, and Mohamed Wiem Mkaouer. Multi-criteria web services selection: Balancing the quality of design and quality of service. *ACM Transactions on Internet Technology (TOIT)*, 22(1):1–31, 2021.

[54] Nuri Almarimi, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. csdetector: an open source tool for community smells detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1560–1564, 2021.

[55] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Bf-detector: an automated tool for ci build failure detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1530–1534, 2021.

[56] Oumayma Hamdi, Ali Ouni, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. A longitudinal study of the impact of refactoring in android applications. *Information and Software Technology*, 140:106699, 2021.

[57] Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. An empirical study on the im-

pact of refactoring on quality metrics in android applications. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 28–39. IEEE, 2021.

[58] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Fabio Palomba. On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Information and Software Technology*, 138:106618, 2021.

[59] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, 135:106566, 2021.

[60] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. On the classification of bug reports to improve bug localization. *Soft Computing*, 25(11):7307–7323, 2021.

[61] Makram Soui, Mabrouka Chouchane, Narjes Bessghaier, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the impact of aesthetic defects on the maintainability of mobile graphical user interfaces: An empirical study. *Information Systems Frontiers*, pages 1–18, 2021.

[62] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021*

IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 348–357. IEEE, 2021.

[63] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. Anti-patterns in modern code review: Symptoms and prevalence. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 531–535. IEEE, 2021.

[64] Xin Ye, Yongjie Zheng, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. Recommending pull request reviewers based on code changes. *Soft Computing*, 25(7):5619–5632, 2021.

[65] Hussein Alrubaye, Deema Alshoaibi, Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. How does library migration impact software quality and comprehension? an empirical study. In *International Conference on Software and Software Reuse*, pages 245–260. Springer, 2020.

[66] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing*, 100:106908, 2021.

[67] Moataz Chouchen, Ali Ouni, and Mohamed Wiem Mkaouer. Androlib: Third-party software library recommendation for android applications.

In *International Conference on Software and Software Reuse*, pages 208–225. Springer, 2020.

[68] Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. On the detection of community smells using genetic programming-based ensemble classifier chain. In *Proceedings of the 15th International Conference on Global Software Engineering*, pages 43–54, 2020.

[69] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821, 2021.

[70] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Web service api anti-patterns detection as a multi-label learning problem. In *International Conference on Web Services*, pages 114–132. Springer, 2020.

[71] Bader Alkhazi, Andrew DiStasi, Wajdi Aljedaani, Hussein Alrubaye, Xin Ye, and Mohamed Wiem Mkaouer. Learning to rank developers for bug report assignment. *Applied Soft Computing*, 95:106667, 2020.

[72] Motaz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Recommending peer reviewers in modern code review: a multi-objective search-based approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 307–308, 2020.

[73] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.

[74] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[75] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. On the prediction of continuous integration build failures using search-based software engineering. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 313–314, 2020.

[76] Nuri Almarimi, Ali Ouni, and Mohamed Wiem Mkaouer. Learning to detect community smells in open source software projects. *Knowledge-Based Systems*, 204:106201, 2020.

[77] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolutionary search. In *International Conference on Service-Oriented Computing*, pages 58–63. Springer, Cham, 2019.

[78] Nuri Almarimi, Ali Ouni, Salah Bouktif, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Mohamed Aymen Saied. Web service api

recommendation for automated mashup creation using multi-objective evolutionary search. *Applied Soft Computing*, 85:105830, 2019.

[79] Makram Soui, Mabrouka Chouchane, Mohamed Wiem Mkaouer, Marouane Kessentini, and Khaled Ghedira. Assessing the quality of mobile graphical user interfaces using multi-objective optimization. *Soft Computing*, 24(10):7685–7714, 2020.

[80] Nasir Safdari, Hussein Alrubaye, Wajdi Aljedaani, Bladimir Baez Baez, Andrew DiStasi, and Mohamed Wiem Mkaouer. Learning to rank faulty source files for dependent bug reports. In *Big Data: Learning, Analytics, and Applications*, volume 10989, page 109890B. International Society for Optics and Photonics, 2019.

[81] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ocinneide, Ali Ouni, and Yuanfang Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, 2018.

[82] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33. ACM, 2018.

[83] Makram Soui, Mabrouka Chouchane, Ines Gasmi, and Mohamed Wiem Mkaouer. Plain: Plugin for predicting the usability of mobile user interface. In *VISIGRAPP (1: GRAPP)*, pages 127–136, 2017.

[84] Ian Shoenberger, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the use of smelly examples to detect code smells in javascript. In *European Conference on the Applications of Evolutionary Computation*, pages 20–34. Springer, Cham, 2017.

[85] Mohamed Wiem Mkaouer. Interactive code smells detection: An initial investigation. In *International Symposium on Search Based Software Engineering*, pages 281–287. Springer, Cham, 2016.

[86] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, and Mel Ó Cinnéide. A robust multi-objective approach for software refactoring under uncertainty. In *International Symposium on Search Based Software Engineering*, pages 168–183. Springer, Cham, 2014.

[87] Mohamed Wiem Mkaouer and Marouane Kessentini. Model transformation using multiobjective optimization. In *Advances in Computers*, volume 92, pages 161–202. Elsevier, 2014.

[88] Mohamed W Mkaouer, Marouane Kessentini, Slim Bechikh, and Daniel R Tauritz. Preference-based multi-objective software modelling. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 61–66. IEEE, 2013.

[89] Taryn Takebayashi, Anthony Peruma, Mohamed Wiem Mkaouer, and Christian D Newman. An exploratory study on the usage and readability of messages within assertion methods of test cases. *arXiv preprint arXiv:2303.00169*, 2023.

[90] Eman Abdullah AlOmar, Salma Abdullah AlOmar, and Mohamed Wiem Mkaouer. On the use of static analysis to engage students with software quality improvement: An experience with pmd. *arXiv preprint arXiv:2302.05554*, 2023.

[91] Wajdi Aljedaani, Mohammed Alkahtani, Stephanie Ludi, Mohamed Wiem Mkaouer, Marcelo M Eler, Marouane Kessentini, and Ali Ouni. The state of accessibility in blackboard: Survey and user reviews case study. In *20th International Web for All Conference*, pages 84–95, 2023.

[92] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Anthony Peruma, and Stephanie Ludi. Do the test smells assertion roulette and eager test impact students' troubleshooting and debugging capabilities? *arXiv preprint arXiv:2303.04234*, 2023.

[93] Anthony Peruma, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. Refactoring debt: myth or reality? an exploratory study on the relationship between technical debt and refactoring. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 127–131, 2022.

[94] Eman Abdullah AlOmar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. Anticopypaster: extracting code duplicates as soon as they are introduced in the ide. In *Proceedings*

of the 37th IEEE/ACM International Conference on Automated Software Engineering, pages 1–4, 2022.

[95] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Automating source code refactoring in the classroom. *arXiv preprint arXiv:2311.10753*, 2023.

[96] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. How is software reuse discussed in stack overflow? *arXiv preprint arXiv:2311.00256*, 2023.

[97] Eman Abdullah AlOmar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. Just-in-time code duplicates extraction. *Information and Software Technology*, 158:107169, 2023.

[98] Eman Abdullah AlOmar, Moataz Chouchen, Mohamed Wiem Mkaouer, and Ali Ouni. Code review practices for refactoring changes: An empirical study on openstack. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 689–701, 2022.

[99] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. An exploratory study on refactoring documentation in issues handling. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 107–111, 2022.

[100] Eman Abdullah AlOmar, Salma Abdullah AlOmar, and Mohamed Wiem Mkaouer. On the use of static analysis to engage students with software quality improvement: An experience with pmd. *arXiv preprint arXiv:2302.05554*, 2023.

[101] Wajdi Aljedaani, Mona Aljedaani, Mohamed Wiem Mkaouer, and Stephanie Ludi. Teachers perspectives on transition to online teaching deaf and hard-of-hearing students during the covid-19 pandemic: A case study. In *Proceedings of the 16th Innovations in Software Engineering Conference*, pages 1–10, 2023.

[102] Deema Adeeb Al Shoaibi and Mohamed Wiem Mkaouer. Understanding software performance challenges an empirical study on stack overflow. In *2023 International Conference on Code Quality (ICCQ)*, pages 1–15. IEEE, 2023.

[103] Waleed Alhindi, Abdulrahman Aleid, Ilyes Jenhani, and Mohamed Wiem Mkaouer. Issue-labeler: an albert-based jira plugin for issue classification. In *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 40–43. IEEE, 2023.

[104] Marwa Daaji, Ali Ouni, Mohamed Mohsen Gammoudi, Salah Bouktif, and Mohamed Wiem Mkaouer. Bpel process defects prediction using multi-objective evolutionary search. *Journal of Systems and Software*, page 111767, 2023.

[105] Ali Ouni, Islem Saidani, Eman Alomar, and Mohamed Wiem Mkaouer. An empirical study on continuous integration trends, topics and challenges in stack overflow. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 141–151, 2023.

[106] Moataz Chouchen, Ali Ouni, Jefferson Olongo, and Mohamed Wiem Mkaouer. Learning to predict code review completion time in modern code review. *Empirical Software Engineering*, 28(4):82, 2023.

[107] Ali Ouni, Eman Abdullah AlOmar, Oumayma Hamdi, Mel Ó Cinnéide, Mohamed Wiem Mkaouer, and Mohamed Aymen Saied. On the impact of single and co-occurrent refactorings on quality attributes in android applications. *Journal of Systems and Software*, 205:111817, 2023.

[108] Wajdi Aljedaani, Mohammed Alkahtani, Stephanie Ludi, Mohamed Wiem Mkaouer, Marcelo M Eler, Marouane Kessentini, and Ali Ouni. The state of accessibility in blackboard: Survey and user reviews case study. In *Proceedings of the 20th International Web for All Conference*, pages 84–95, 2023.

[109] Wajdi Aljedaani, Rrezarta Krasniqi, Sanaa Aljedaani, Mohamed Wiem Mkaouer, Stephanie Ludi, and Khaled Al-Raddah. If online learning works for you, what about deaf students? emerging challenges of online learning for deaf and hearing-impaired students during covid-19: a literature review. *Universal access in the information society*, 22(3):1027–1046, 2023.

[110] Deema Alshoaibi, Ikram Chaabane, Kevin Hannigan, Ali Ouni, and Mohamed Wiem Mkaouer. On the detection of performance regression introducing code changes: Experience from the git project. In *2022 IEEE 29th Annual Software Technology Conference (STC)*, pages 206–217. IEEE, 2022.

[111] Wajdi Aljedaani, Furqan Rustam, Mohamed Wiem Mkaouer, Abdullatif Ghallab, Vaibhav Rupapara, Patrick Bernard Washington, Ernesto Lee, and Imran Ashraf. Sentiment analysis on twitter data integrating textblob and deep learning models: The case of us airline industry. *Knowledge-Based Systems*, 255:109780, 2022.

[112] Wajdi Aljedaani, Ibrahem Abuhaimed, Furqan Rustam, Mohamed Wiem Mkaouer, Ali Ouni, and Ilyes Jenhani. Automatically detecting and understanding the perception of covid-19 vaccination: a middle east case study. *Social Network Analysis and Mining*, 12(1):128, 2022.

[113] Eman Abdullah AlOmar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. Just-in-time code duplicates extraction. *Information and Software Technology*, 158:107169, 2023.

[114] Deema ALShoaibi, Hiten Gupta, Max Mendelson, Ilyes Jenhani, Ali Ben Mrad, and Mohamed Wiem Mkaouer. Learning to characterize performance regression introducing code changes. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1590–1597, 2022.

[115] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Stephanie Ludi, Ali Ouni, and Ilyes Jenhani. On the identification of accessibility bug reports in open source systems. In *Proceedings of the 19th international web for all conference*, pages 1–11, 2022.

[116] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1):21, 2022.

[117] Deema Alshoaibi, Mohamed Wiem Mkaouer, Ali Ouni, AbdulMutalib Wahaishi, Travis Desell, and Makram Soui. Search-based detection of code changes introducing performance regression. *Swarm and Evolutionary Computation*, 73:101101, 2022.

[118] Behrouz Zolfaghari, Reza Parizi, Gautam Srivastava, and Yoseph Hailemariam. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software Practice and Experience*, 51, 11 2020.

[119] J. Micco. Flaky tests at google and how we mitigate them. *Online*, 05 2016.

[120] J. Palmer. Test flakiness – methods for identifying and dealing with flaky tests. *Online*, 05 2019.

[121] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 312–322, 2019.

[122] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. Ifixflakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 545–555, New York, NY, USA, 2019. Association for Computing Machinery.

[123] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. An evaluation of machine learning methods for predicting flaky tests. In *QuASoQ@APSEC*, 2020.

[124] Guillaume Haben, Sarra Habchi, Mike Papadakis, Maxime Cordy, and Yves Le Traon. A replication study on the usability of code vocabulary in predicting flaky tests. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 219–229, 2021.

[125] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. Flakeflagger: Predicting flakiness without rerunning tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1572–1584, 2021.

[126] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Trans. Softw. Eng. Methodol.*, 31(1), oct 2021.

[127] Anthony Peruma, Khalid Almalki, Christian Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: an open source test

smells detection tool. pages 1650–1654, 11 2020.