Exception Miner: Multi-language Static Analysis Tool to Identify Exception Handling Anti-Patterns

Jairo Souza*
Tales Alves*
jrmcs@cin.ufpe.br
tvac@cin.ufpe.br
Federal University of Pernambuco
Recife, Pernambuco, Brazil

Robson Oliveira
Leopoldo Teixeira
ropj@cin.ufpe.br
lmt@cin.ufpe.br
Federal University of Pernambuco
Recife, Pernambuco, Brazil

Baldoino Fonseca
baldoino@ic.ufal.br
Federal University of Alagoas
Maceió, alagoas, Brazil

ABSTRACT

Exception handling is a technique used to manage errors or exceptional events that occur during program execution and disrupt the standard flow of the program. Although this method is beneficial, developers often neglect proper exception handling. Misuse of exception handling mechanisms can lead to exception handling anti-patterns in the source code, negatively impacting software quality. Additionally, many projects nowadays are built using multiple languages, which increases the complexity of identifying these anti-patterns and the workload involved in configuring various linters or detection tools. To address this gap, we developed a tool that identifies exception handling anti-patterns in three popular languages: Java, TypeScript, and Python. To evaluate our tool, we conducted an empirical study on 11 multi-language projects to investigate the occurrence of exception handling patterns across different languages. Our results demonstrated that the occurrence of exception handling anti-patterns is similar between Python and TypeScript. However, in Java, exception handling anti-patterns occur up to four times more frequently in the analyzed projects. The tool is available on GitHub[1], along with a video[2] demonstrating its use.

KEYWORDS

Exception Handling, Anti-Patterns, Mining Software Repositories, Static Analysis

1 INTRODUCTION

Exception handling is a crucial aspect of programming, involving techniques to manage errors or unexpected events that can interrupt the normal execution flow of a program [2]. These techniques, known as "*exception handling mechanisms*", are commonly found in many programming languages, including Java, TypeScript, and Python [7]. Widely used in various projects, from enterprise solutions to open-source development, effective exception handling enhances software quality by improving code readability, reliability, and maintainability [17].

Previous studies have revealed the widespread presence of exception handling in systems and their relation to anti-patterns in different languages. Most research analyzing exception handling anti-patterns predominantly focuses on Java [4, 8, 15], leaving a gap in comprehensive multi-language analysis. These studies have shown that addressing these aspects is crucial for improving software quality. However, other studies [6, 13] have indicated that developers often neglect best practices in exception handling, underestimate the impact of exception handling anti-patterns, and consequently introduce them during the software development.

This study aims to develop and evaluate Exception Miner, specifically designed to identify exception handling anti-patterns in three programming languages: Java, TypeScript, and Python. This approach aims to unify the detection process, allowing developers to avoid the complexity and unnecessary work of using multiple linters or static analysis tools with different configurations and outputs for each language. Using Exception Miner, it is possible to detect various exception handling anti-patterns in a multi-language project. To achieve this, the tool mines GitHub repositories and detects exception handling anti-patterns across these three languages.

Moreover, we evaluated our tool by mining 11 open source repositories containing at least two languages. Our results demonstrate that all multi-language open-source projects contain exception handling anti-patterns in at least one language. Notably, Java exhibits more anti-patterns than Python and TypeScript, with *Exception Swallowing* and *Destructive Wrapping* being the most frequent, occurring 2,286 and 1,196 times, respectively. Our tool enables practitioners and researchers to detect various types of exception handling anti-patterns in multi-language projects within their CI pipelines or for conducting empirical studies.

2 BACKGROUND

Exception handling is a critical aspect of software development that deals with unexpected events (errors) during a program's execution. Its primary purpose is to ensure the robustness and reliability of software systems, address exceptional conditions, prevent abrupt program termination, and enable programs to respond or recover appropriately from unexpected situations [1, 7]. Handling an exception depends on whether the code can and should recover as expected. If recovery is not possible, it is best to let the error propagate through the execution flow [1]. For example, in Python, the control flow is defined by a set of mechanisms (e.g., `except`, `raise`), that enable the developer to determine the path or transfer control whenever an error occurs.

2.1 Common Exception Handling Mechanisms

Python, Java, and TypeScript share the following core mechanisms for exception handling:

*Both authors contributed equally to this research.
[1]https://github.com/RobsonOlv/exception-miner
[2]https://tinyurl.com/fjb8sz89

2.1.1 Try-Catch (Try-Except in Python). Used to catch exceptions that occur in the `try` block and handle them in the `catch` (or `except` in Python) block.

- **Python**:

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error occurred: {e}")
```

- **Java**:

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Error occurred: " +
                       e.getMessage());
}
```

- **TypeScript**:

```
try {
    let result = 10 / 0;
} catch (e) {
    console.log("Error occurred: " + e.message);
}
```

2.1.2 Try-Finally. Ensures that the `finally` block executes regardless of whether an exception was thrown, typically used for resource cleanup.

- **Python**:

```
try:
    result = 10 / 0
finally:
    print("This will always run.")
```

- **Java**:

```
try {
    int result = 10 / 0;
} finally {
    System.out.println("This will always run.");
}
```

- **TypeScript**:

```
try {
    let result = 10 / 0;
} finally {
    console.log("This will always run.");
}
```

2.1.3 Throwing/Raising Exceptions. Exceptions can be explicitly thrown (or raised) using the `throw` (or `raise` in Python) statement.

Example:

- **Python**:

```
raise ValueError("An error occurred")
```

- **Java**:

```
throw new RuntimeException("An error occurred");
```

- **TypeScript**:

```
throw new Error("An error occurred");
```

2.2 Language-Specific Constructs

In addition to these common constructs, Python provides some unique mechanisms:

2.2.1 Try-Except-Else. Executes the `else` block if the code inside the `try` block does not raise an exception.

Example:

```
try:
    result = 10 / 2
except ZeroDivisionError as e:
    print(f"Error occurred: {e}")
else:
    print("No errors occurred, result is:", result)
```

2.3 Exception Handling Anti-Patterns

Anti-patterns are common mistakes developers make during software development [11], while exception handling anti-patterns are specifically associated with exception handling mechanisms. Previous studies emphasize the negative impact of improper practices, demonstrating how these anti-patterns can introduce bugs and compromise code maintainability [10, 14, 15]. These studies suggest that such anti-patterns should be avoided when developing or evolving software. Below, we describe the exception handling anti-patterns used in our study. We selected these exception handling anti-patterns based on previous studies [5, 9, 10, 12] and current static analysis tools or linters.

Exception Handling Anti-Patterns in Python:

- **Too Broad Raising**: Raising exceptions that are too general, lacking specificity. For example, using a *Generic* exception type. This can hide bugs and make debugging harder.
- **Too Broad Except**: Handling exceptions that are too general, lacking specificity.
- **Swallowing Exceptions**: Doing nothing or logging insufficient information inside an exception handler potentially leads to silent errors. This exception handling anti-pattern is common in Python using the `try-except-pass` flow.
- **Nested Try-Except Blocks**: Using three or more nested try-except blocks, indicating complex exception handling logic.
- **Bare Raise Block:** A `raise` statement with no exception provided, i.e., re-raising the last active exception. Bare raises should only be used in `except` blocks;
- **Bare Except Catch Block**: An `except` block with no exception provided. Bare except catches all exceptions, including exceptions such as `SystemExit` or `KeyboardInterrupt`.
- **Unhandled Exception**: Occurs when the application does not properly handle exceptions, leading to unexpected program termination or errors.

Table 1: Summary of Exception Handling Mechanisms

Mechanism	Python	Java	TypeScript
Try-Catch (Except)	`try ... except`	`try ... catch`	`try ... catch`
Try-Finally	`try ... finally`	`try ... finally`	`try ... finally`
Throw/Raise	`raise`	`throw`	`throw`
Try-Except-Else	`try ... except ... else`	Not applicable	Not applicable

- **Bare Raise inside Finally**: Raising exceptions directly within a `finally` block, which may lead to unexpected states and conditions.
- **Try and Return**: Using a `try` block solely to return from a function indicates a potential design issue.

Exception Handling Anti-Patterns in Typescript:

- **Empty Catch**: Using the empty `catch` block, which means catching an exception but not handling it.
- **Wrapped Catch**: Do not use the error thrown by `catch` block. This means losing the error and its traceability, which are key to identifying the problem and its origin, as well as generating useful feedback
- **Throw Literal**: Throwing an exception without using any standard JavaScript error types derived from the JavaScript *Error* object. We can use the stack trace for the error when we throw an exception using an Error object.
- **Useless Catch**: Blocks that only re-throw the exception, equivalent to not handling the error.
- **Error Reassignment**: Reassigning the captured error in the `catch` block is potentially problematic as it can lose the traceability of the original error.

Exception Handling Anti-Patterns in Java:

- **Throwing Generic Exception**: Throwing exceptions that are too general, such as `Exception` or `Throwable`, lacks specificity and can obscure the nature of the error.
- **Throwing Raw Exception**: Directly throwing `RuntimeException` or other raw exceptions without additional context or wrapping is not informative.
- **Throw from Within Finally**: Throwing exceptions inside a `finally` block can mask exceptions thrown in the `try` or `catch` blocks.
- **Throwing NullPointerException**: Explicitly throwing `NullPointerException` represents a programmer error and should be avoided.
- **Handling Generic Exception**: Catching exceptions that are too general, such as `Exception` or `Throwable`, can obscure the specific nature of the error.
- **Relying on `instanceof` in Catch Blocks**: Using `instanceof` to check the type of an exception in a generic catch block indicates poor exception hierarchy design.
- **Nesting Try-Catch Blocks More than Twice**: Using three or more nested try-catch blocks indicates a complex and hard-to-maintain exception handling mechanism.
- **Exception Swallowing**: Catching exceptions, doing nothing, or logging insufficient information can lead to silent failures.
- **Destructive Wrapping/Logging**: Wrapping exceptions without preserving the original exception's context, stack trace, or logging exceptions without enough detail.
- **Masking Programmer Errors**: Using exception handling to mask errors that should be fixed by the programmer, such as incorrect logic or null pointer references.

3 EXCEPTION MINER TOOL

We introduce Exception Miner, a multi-language static analysis tool designed to identify exception handling anti-patterns. The tool is designed to provide a command-line interface for analyzing Python, Java, and TypeScript projects.

3.1 Comparison of Related Tools

In Table 2, we compare existing tools that address code quality issues, including those related to exception handling. While these tools may not specifically focus on identifying exception handling anti-patterns across multiple languages, they provide valuable insights into the landscape of static analysis and code quality tools.

These tools offer valuable capabilities in detecting code quality issues, but none comprehensively cover the automated analysis of exception handling anti-patterns across multiple programming languages. Our Exception Miner fills this gap by providing a unified solution for identifying and addressing exception handling anti-patterns in Java, TypeScript, and Python source code.

3.2 Exception Miner Architecture

The Exception Miner architecture is designed to facilitate the detection of exception handling anti-patterns across multiple programming languages. The method involves five main stages, each contributing to the comprehensive analysis of the selected repositories. An overview of the tool's architecture is illustrated in Figure 1.

The architecture has five main components: CLI Interface, Repository Manager, File Analyzer, Anti-Pattern Detector, and Data Aggregator. Each is responsible for specific tasks in the analysis process.

1. CLI Interface: The CLI Interface is the entry point for users to interact with the tool. Users specify the parameters for the analysis, such as the list of repository links, the programming languages to be analyzed, and other configuration settings. This component is responsible for:

- Collecting repository links.
- Accepting user-specified languages for analysis.

2. Repository Manager: The Repository Manager handles downloading and managing the repositories specified by the user. It interacts with the GitHub API to fetch the required repositories. This component performs the following steps:

Tool	Purpose	Supported Languages	Features
FindBugs	Static analysis for Java	Java	Detects potential bugs, including issues in exception handling.
PMD	Static analysis for Java	Java	Identifies common coding flaws, including issues related to exception handling.
ESLint	Static analysis for JavaScript	JavaScript	Identifies coding errors and style inconsistencies, which may include problematic exception handling mechanisms.
PyLint	Static analysis for Python	Python	Checks for errors, style issues, and anti-patterns, including potential issues in exception handling.
SonarQube	Continuous inspection platform	Multiple languages	Provides static analysis rules covering various programming languages such as Java, JavaScript, and Python.

Table 2: Comparison of Related Tools

- Sending HTTP requests to the GitHub API.
- Downloading projects from GitHub.
- Organizing the downloaded projects for further analysis.

3. File Analyzer: The File Analyzer examines each file within the downloaded projects to identify those written in the specified programming languages. It uses the tree-sitter library[3] to parse and analyze the syntax of the code files. This component is responsible for:

- Parsing the code to understand its structure.
- Identifying files based on the user-specified languages.

4. Anti-Pattern Detector: The Anti-Pattern Detector is the core component that analyzes the parsed code to identify exception handling anti-patterns. It applies specific AST (Abstract Syntax Tree) queries to detect various anti-patterns in Python, Java, and TypeScript code. This component performs the following tasks:

- Applying AST queries to detect anti-patterns.
- Identifying exception handling anti-patterns in the analyzed code.

5. Data Aggregator: The Data Aggregator compiles the analysis results and creates data frames for each project. These data frames contain detailed information about the detected exception handling anti-patterns. The tasks of this component include:

- Aggregating the detected anti-patterns data.
- Creating data frames for each analyzed project.

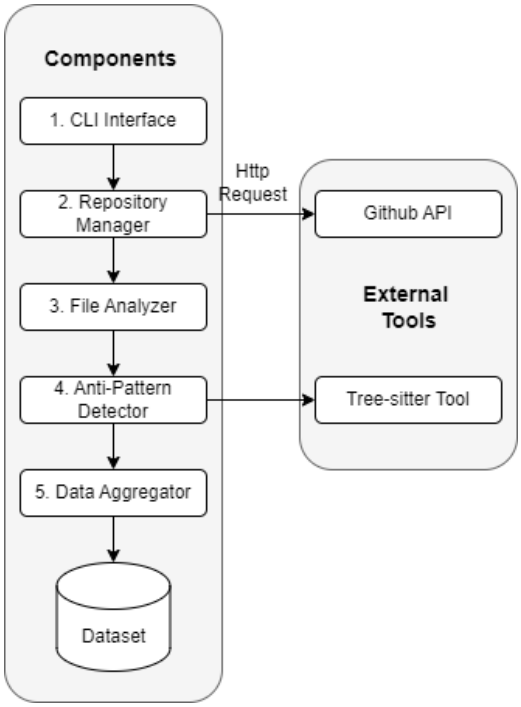

Figure 1: Overview of the Exception Miner Architecture.

4 STUDY DESIGN

To demonstrate and evaluate our tool, we focus on answering our research question: *How common are exception handling anti-patterns in open-source projects?* With this, we pretend to investigate the presence of exception handling anti-patterns in open-source projects. Such analysis provides an overview of each type of exception handling anti-pattern across different projects and programming languages. This information can help researchers and developers to focus their efforts on exception handling anti-patterns that occur most frequently in the multi-language projects analyzed in our study.

4.1 Selected Projects

To perform our study, we use the SEART GitHub Search Engine [3][4] to identify relevant multi-language open-source projects on GitHub. We select projects based on previous studies [6, 10] and by considering the following criteria:

- the projects need to be open-source, hosted on GitHub, and developed in Python, Typescript, and/or Java since access to closed software systems is usually limited;
- The projects must be active with at least one commit in the last three months and maintained for at least three years. The main motivation is to collect relevant evolving and long-term projects;

[3]https://tree-sitter.github.io/tree-sitter/

[4]https://seart-ghs.si.usi.ch/

- the projects must have relevant popularity based on contributors and stars, so we only consider projects with more than ten contributors and 1k stars indicating developers' community engagement;
- The projects need to have at least 2 languages.

We select projects with distinct characteristics, such as commits, issues, and size, based on the aforementioned criteria. Table 3 describes the list of selected projects in our study, presenting their number of commits, number of developers, popularity based on contributors, stars, and programming languages. As a result, our study analyzes a total of 11 multi-language projects, characterized by a total of 3.0050 contributors, 142.8k stars, and 84.211 commits.

Table 3: An overview of the selected multi-language projects.

Project	# Contributors	# Stars	# Commits	Languages
Arduino	238	14k	7.375	Java, Python
Airbyte	908	14.5k	16.103	Java, Python
Apitable	66	12.2k	917	Typescript, Java
Ar-cutpaste	6	14.6k	50	Typescript, Python
Capacitor	273	11.3k	4.296	Typescript, Java
Flipper	298	13.2k	9.808	Typescript, Java
Gitpod	135	12.4k	11.695	Typescript, Java
Jupyterlab	543	13.9k	26.020	Typescript, Python
Kubeflow	296	13.8k	2.556	Typescript, Python
Leon	18	14.8k	1.614	Typescript, Python
Python-for-android	224	8.1k	3.777	Typescript, Java
Total:	3.005	142.8k	84.211	-

4.2 Collecting Exception Handling Anti-Patterns

This section describes how our tool collects and identifies exception handling anti-patterns across various projects (The exception handling anti-patterns collected are demonstrated in Section 2.3). To perform that, we developed a specific Abstract Syntax Tree (AST) query using the tree-sitter library to detect these patterns in Python, Java, or JavaScript code for each identified anti-pattern.

For example, consider the "Nested Try-Except" anti-pattern, initially identified and cataloged in a previous study [12]. We translated the manual identification of this anti-pattern into an AST query that our tool uses to automate the detection process. This translation enables the systematic identification of such anti-patterns across all the analyzed projects. To illustrate this, we formulated an AST query for the "Nested Try-Except" exception handling anti-pattern using the tree-sitter library to identify try statements:

```
QUERY_TRY_STMT: Query = PY_LANGUAGE.query(
    """(try_statement) @try.statement""")
```

Then, to detect nested try-except blocks, we define a function that works as follows:

(1) It captures all try statements within a given AST node.
(2) It then iterates through these captures to check if any try statement contains more than two nested try statements using the previous query.
(3) If such a nested structure is found, it returns `True`, flagging the presence of the "Nested Try-Except" anti-pattern.

Our tool systematically identifies and catalogs exception handling anti-patterns across various programming languages and projects by employing such AST queries and detection functions.

5 RESULTS

Our study aims to identify which exception handling anti-patterns are more common in the 11 open-source multi-language projects analyzed in our study. First, we present a general analysis of the occurrence of all exception handling anti-patterns across all projects. Furthermore, we analyze the occurrence of each exception handling anti-patterns in the respective programming languages: Python, Java, and Typescript.

Table 4 describes the results of each programming language across all projects. The first column presents the project, and the remaining column describes the number of exception handling anti-patterns occurrence and their respective percentages across the programming languages.

Table 4: Distribution of exception handling anti-patterns in Python, Typescript and Java.

Project	Python	Typescript	Java
Arduino	24 (2.07%)	0 (0.00%)	1133 (97.93%)
Airbyte	760 (31.43%)	0 (0.00%)	1658 (68.57%)
Apitable	0 (0.00%)	221 (26.09%)	626 (73.91%)
Ar-cutpaste	0 (0.00%)	2 (100.00%)	0 (0.00%)
Capacitor	0 (0.00%)	135 (24.37%)	419 (75.63%)
Flipper	0 (0.00%)	348 (57.90%)	253 (42.10%)
Gitpod	0 (0.00%)	230 (27.54%)	605 (72.46%)
Jupyterlab	48 (19.20%)	202 (80.80%)	0 (0.00%)
Kubeflow	9 (50.00%)	9 (50.00%)	0 (0.00%)
Leon	39 (90.70%)	4 (9.30%)	0 (0.00%)
Python-for-android	44 (21.05%)	0 (0.00%)	165 (78.95%)
Total	**924 (13.33%)**	**1.151 (16.60%)**	**4.859 (70.07%)**

The results show that the total number of exception handling anti-patterns is 924 (13.33%) in Python projects, 1151 (16.60%) in TypeScript, and 4.859 (70.07%) in Java. The Arduino project exhibits the highest number of exception handling anti-patterns in Java with 1.133 exception handling anti-patterns, while the Airbyte project contains the highest number in Python, reaching 760 exception handling anti-patterns in Airbyte. Finally, TypeScript language contains the distribution of exception handling anti-patterns with values between 0 and 348 (accounting for 57.90% of the exception handling anti-patterns from Flipper project). These results suggest that exception handling anti-patterns in Java occur more frequently than Python and Typescript.

To provide a detailed view, Table 5 presents the results of exception handling anti-patterns regarding Java programming language in the projects analyzed. The first column lists the project, while the subsequent columns show the occurrences of five specific anti-patterns: *Handling Generic*, *Throwing Generic*, *Nested Try*, *Wrapped Catch*, and *Exception Swallowing*.

The results demonstrated a higher occurrence of the exception handling anti-patterns *Destructive Wrapping* and *Exception Swallowing*. For example, the Arduino project contains 291 exception handling anti-patterns occurrences of *Destructive Wrapping* and 572 occurrences of *Exception Swallowing*. Similarly, the Airbyte project has 405 *Destructive Wrapping* and 782 *Exception Swallowing* occurrences. These two anti-patterns are the most frequent across all projects, totaling 1.196 and 2.286 occurrences, respectively.

Table 5: Distribution of exception handling anti-patterns in Java

projeto	Handling Generic	Throwing Generic	Nested Try	Destructive Wrapping	Exception Swallowing
Arduino	126	1	2	291	572
airbyte	148	0	0	405	782
apitable	56	3	2	160	302
ar-cutpaste	0	0	0	0	0
capacitor	39	0	0	115	220
flipper	20	0	3	72	128
gitpod	1	0	0	96	192
jupyterlab	0	0	0	0	0
kubeflow	0	0	0	0	0
leon	0	0	0	0	0
python-for-android	10	0	0	57	90
Total	**400**	**4**	**7**	**1196**	**2286**

In contrast, the remaining exception handling anti-patterns are less common. For example, *Handling Generic* occurs 400 times. Additionally, *Nested Try* is relatively rare, with only 7 occurrences across all projects, and *Throwing Generic* is the least frequent, with just 4 occurrences. Moreover, the Ar-cutpaste, Jupyterlab, Kubeflow, and Leon projects show no exception handling anti-patterns, indicating better quality in their exception handling mechanisms.**In summary, our results highlight a significant disparity among different anti-patterns, with *Destructive Wrapping* and *Exception Swallowing* being the most prevalent in the multi-language open-source projects analyzed**. Additional results regarding the exception handling anti-patterns in Python and TypeScript are available on the accompanying web page [16].

6 TOOL IMPACTS AND LIMITATIONS

Exception Miner is a mining tool that parses and analyzes source code written in Python, Java, and TypeScript, making it a valuable resource for developers in projects with diverse programming languages. The tool features a user-friendly command-line interface (CLI) that facilitates the integration into existing workflows (CI/CD platforms). By focusing on collecting and reporting exception handling anti-patterns, our tool provides detailed insights about potential issues that could compromise code quality and maintainability.

For developers, Exception Miner offers significant advantages by unifying the detection of exception handling anti-patterns across different languages. This eliminates the need for multiple linters, each with different outputs and interfaces, simplifying maintaining high code standards in multilingual projects. Developers can use this tool to identify poor practices of exception handling mechanisms to improve the software quality, leading to more robust and reliable software. Also, it helps developers save time and effort, enabling them to focus on writing efficient code without worrying about exception handling mechanisms language-specific nuances. Moreover, our tool allows researchers to conduct empirical studies using exception handling mechanisms across various languages. Researchers can use its cross-language analysis capabilities to mine software repositories and provide an extensive output (dataset) on how exception handling mechanisms are used in real-world projects. Thus, this output can help us better understand the use of exception handling mechanisms.

While Exception Miner offers robust capabilities for analyzing exception handling mechanisms in Python, Java, and TypeScript, it does have some limitations. Notably, the tool does not support all main programming languages, which may restrict its applicability in projects incorporating these languages. Additionally, the tool's effectiveness relies on predefined anti-pattern definitions using AST queries as demonstrated in Section 4.2, which might not cover every possible nuance of exception handling anti-patterns. Finally, the tool does not cover all possible exception handling anti-patterns available in the literature, since as a static analysis tool, Exception Miner cannot detect runtime-specific issues or dynamically generated exception handling anti-patterns, potentially not identifying some exception handling anti-patterns that only manifest during execution (e.g., uncaught exceptions).

7 CONCLUSION AND FUTURE WORK

This paper demonstrated the Exception Miner tool for identifying exception handling anti-patterns in three programming languages: Java, Python, and TypeScript, to support software maintenance. The tool aims to enhance software quality by identifying exception handling anti-patterns in these popular languages. Our results indicate that all multi-language open-source projects contain exception handling anti-patterns in at least one language. Notably, Java exhibits more exception handling anti-patterns than Python and TypeScript, with *Exception Swallowing* and *Destructive Wrapping* being the most frequent, with 2,286 and 1,196 occurrences, respectively. Future work will involve evaluating our tool in additional open-source projects and other programming languages.

AVAILABILITY OF ARTIFACTS

The complete results (from Python and JavaScript), along with the Exception Miner tool, and the link to the video, are available on the accompanying web page. [16].

REFERENCES

[1] David M. Beazley. 2022. *Python distilled.* Addison-Wesley.

[2] Joshua Bloch. 2008. *Effective Java™, Second Edition* (second ed.). Prentice Hall Press, USA.

[3] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021.* IEEE, 560–564.

[4] C. Hsieh, C. L.My, Kim T. Ho, and Yu C. Cheng. 2017. Identification and Refactoring of Exception Handling Code Smells in JavaScript. *Journal of Internet Technology* 18, 6 (2017), 1461–1471.

[5] Chin-Yun Hsieh, Canh Le My, Kim Thoa Ho, and Yu Chin Cheng. 2017. Identification and Refactoring of Exception Handling Code Smells in JavaScript. *Journal of Internet Technology* 18, 6 (2017), 1461–1471. https://jit.ndhu.edu.tw/article/view/1597

[6] Miryung Kim, Romain Robbes, Christian Bird, Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the exception handling strategies of Java libraries. *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), 212–222. https://doi.org/10.1145/2901739.2901757

[7] Oracle. 2014. What is an exception? https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

[8] Haidar Osman, Andrei Chiş, Claudio Corrodi, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Exception Evolution in Long-lived Java Systems. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (2017), 302–311. https://doi.org/10.1109/msr.2017.21

[9] Yun Peng, Yu Zhang, and Mingzhe Hu. 2021. An Empirical Study for Common Language Features Used in Python Projects. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* 00 (2021), 24–35. https://doi.org/10.1109/saner50967.2021.00012

[10] Guilherme Bicalho de Pádua and Weiyi Shang. 2017. Studying the Prevalence of Exception Handling Anti-Patterns. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)* (2017), 328–331. https://doi.org/10.1109/icpc.2017.1

[11] Linda Rising. 1998. *Design patterns: elements of reusable architectures.* Cambridge University Press, USA, 9–17.

[12] Jonathan Rocha, Hugo Melo, Roberta Coelho, and Bruno Sena. 2018. Towards a Catalogue of Java Exception Handling Bad Smells and Refactorings. In *Proceedings*

of the 25th Conference on Pattern Languages of Programs (Portland, Oregon) *(PLoP '18)*. The Hillside Group, USA, Article 7, 17 pages.

[13] Hina Shah, Carsten Görg, and Mary Jean Harrold. 2008. Why do developers neglect exception handling?. In *Proceedings of the 4th International Workshop on Exception Handling* (Atlanta, Georgia) *(WEH '08)*. Association for Computing Machinery, New York, NY, USA, 62–68. https://doi.org/10.1145/1454268.1454277

[14] Dêmora Bruna Cunha de Sousa, Paulo Henrique Maia, Lincoln Souza Rocha, and Windson Viana. 2018. Analysing the Evolution of Exception Handling Anti-Patterns in Large-Scale Projects: A Case Study. *Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse on - SBCARS '18* (2018), 73–82. https://doi.org/10.1145/3267183.3267191

[15] Dêmora B C de Sousa, Paulo Henrique M. Maia, Lincoln S Rocha, and Windson Viana. 2020. Studying the evolution of exception handling anti-patterns in a long-lived large-scale project. *Journal of the Brazilian Computer Society* 26, 1 (2020), 1. https://doi.org/10.1186/s13173-019-0095-5

[16] Jairo Souza, Tales Alves, Robson Oliveira, Leopoldo Texeira, and Baldoino Fonseca. 2024. Complementary Material. https://github.com/RobsonOlv/exception-miner

[17] Veselin Kolev Svetlin Nakov. 2013-09-01. *Fundamentals of Computer Programming with C#*. Faber Publishing.