

Bugs in the Shadows: Static Detection of Faulty Python Refactorings

Jonhnanthan Oliveira

Federal University of Campina Grande
Campina Grande, PB, Brazil
jonhnanthan@copin.ufcg.edu.br

Márcio Ribeiro

Federal University of Alagoas
Maceió, AL, Brazil
marcio@ic.ufal.br

Rohit Gheyi

Federal University of Campina Grande
Campina Grande, PB, Brazil
rohit@dsc.ufcg.edu.br

Alessandro Garcia

Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, RJ, Brazil
afgarcia@inf.puc-rio.br

ABSTRACT

Python is a widely adopted programming language, valued for its simplicity and flexibility. However, its dynamic type system poses significant challenges for automated refactoring – an essential practice in software evolution aimed at improving internal code structure without changing external behavior. Understanding how type errors are introduced during refactoring is crucial, as such errors can compromise software reliability and reduce developer productivity. In this work, we propose a static analysis technique to detect type errors introduced by refactoring implementations for Python. We evaluated our technique on Rope refactoring implementations, applying them to open-source Python projects. Our analysis uncovered 29 bugs across four refactoring types from a total of 1,152 refactoring attempts. Several of these issues were also found in widely used IDEs such as PyCharm and PyDev. All reported bugs were submitted to the respective developers, and some of them were acknowledged and accepted. These results highlight the need to improve the robustness of current Python refactoring tools to ensure the correctness of automated code transformations and support reliable software maintenance.

KEYWORDS

Refactoring, Type error, Testing, Python

1 Introduction

Python is a widely used programming language, valued for its simplicity and flexibility [28]. Its syntax supports multiple paradigms – such as imperative, object-oriented, and functional – and allows developers to choose between typed and untyped code. While these features contribute to Python’s popularity, they also complicate code maintenance and tooling support, especially for automated refactoring [7, 13, 21].

Refactoring is a key practice in software evolution that aims to improve internal code structure without changing its external behavior. Despite its conceptual elegance, implementing correct and reliable refactoring transformations is non-trivial [30, 32, 33, 37, 39, 40]. This task becomes even more challenging in dynamically typed languages such as Python [29], where type-related issues can easily escape compile-time checks and manifest at runtime as errors or unintended behavior.

Prior research has introduced techniques for evaluating the correctness of refactoring tools in languages other than Python. For instance, Schäfer et al. [30] developed an intermediate representation to simplify the specification and verification of Java refactorings. Gligoric et al. [9] tested refactoring implementations in Eclipse for Java and C, uncovering 120 bugs, nearly 30% of which were type-related. Other studies, such as those by Mongiovi et al. [15, 17] and Soares et al. [34, 36], investigated the presence of compilation errors, behavioral changes, and overly strong preconditions in Java-based refactoring engines. In contrast, the extent to which Python refactoring tools introduce type errors remains underexplored. Given the increasing reliance on Integrated Development Environments (IDEs) that support automated refactoring, ensuring the robustness of these tools is crucial. There is an increasing demand for improved tooling support among Python developers [11, 44]. Tools must handle Python’s dynamic typing without introducing type errors, behavioral regressions, or incorrect precondition assumptions [15, 19, 23, 31, 34, 36, 38].

In this paper, we propose a technique in Section 3 to statically detect type errors introduced by refactoring implementations in Python (referred to as `SAFEREFACTORPY`). Our approach applies each refactoring transformation individually to real-world Python projects and uses a differential analysis technique based on Pyre [14], a static type checker for Python, to compare the type-checking results between the original and refactored versions of the code. We evaluate in Section 4 our technique by applying 1,152 transformations from refactorings implemented in Rope [27]. Our technique identified 29 bugs across four refactoring types. Manual evaluation of Rope’s transformations in popular IDEs such as PyCharm, PyDev, and VSCode revealed that some of these bugs also appear in PyCharm and PyDev. We reported all discovered bugs to the relevant developers, and some of them were acknowledged and accepted.

The results of our study have important implications for the development and adoption of automated refactoring tools in dynamically typed languages such as Python. Our findings reveal that even widely used refactoring engines can introduce subtle type-related bugs, which may go unnoticed by developers and result in runtime errors or degraded software quality. This highlights the need for rigorous validation mechanisms in refactoring tools to ensure semantic preservation and correctness of transformations. By statically detecting type errors introduced by refactorings, our

approach can serve as a complementary validation step during tool development or integration in IDEs, ultimately contributing to more reliable code evolution workflows. Furthermore, the acceptance of our bug reports by tool maintainers underscores the practical relevance and real-world applicability of our technique. In summary, our study contributes the following:

- We introduce a static analysis technique to identify type errors in Python refactoring implementations (Section 3);
- We apply the technique to 1,152 transformations, identifying 29 bugs in 4 refactoring implementations (Section 4).

All study artifacts are available online [20].

2 Motivating Example

This section presents a motivating example of a type error introduced by a refactoring in Python, illustrating the kinds of challenges our study seeks to address. Consider a Python program that reads data from a CSV file and compares instances based on an initial value. Listing 1 shows a Python class, `Mark`, which defines functions for setting the initial state¹ and comparing two objects. This example is adapted from the source code of the *TextBlob* project.

```
import csv
class Mark(object):
    def __init__(self, marks, fp):
        self._marks = marks
        reader = csv.reader(
            fp,
            delimiter=';'
        )
        for row in reader:
            print(row)
    def key(self):
        return self._marks
    def __lt__(self, other):
        return self.key() < other.key()
with open('some.csv') as csvfile:
    mark1 = Mark(9, csvfile)
    mark2 = Mark(8, csvfile)
print(mark1 < mark2)
```

Listing 1: Initial Python Program.

Suppose we want to apply the Rename Method refactoring using Rope [27] to change the name of the `__lt__` function in Listing 1 to `compare`, aiming to better reflect its functionality as a comparator of objects based on key values. According to its specification [7], the Rename Method refactoring implementation in Rope updates both the function declaration and all of its references. The resulting program replaces all occurrences of `__lt__` with `compare`.

However, when executing the program, the Python interpreter² reads the CSV file and raises a runtime error. This occurs because `compare` is not a recognized function in Python's rich comparison model³, which relies on specific dunder functions (e.g., `__lt__`, `__gt__`) for object comparisons. In this case, the refactoring implementation introduces an error by renaming a special function that is semantically tied to Python's runtime behavior. This transformation should have been disallowed by a precondition that detects such special functions. This bug is documented in Rope's issue

tracker.⁴ It highlights the need for refactoring tools to include semantic awareness in their precondition checks, especially when dealing with language-specific conventions and reserved function names.

Ideally, such type-related errors should be detected statically by the refactoring engine, rather than being discovered only at runtime through execution errors. This example illustrates a broader challenge stemming from Python's dynamic type system. Refactorings involving renaming functions, fields, or classes may result in runtime errors if all relevant references and semantic constraints are not properly accounted for. Due to the dynamic and implicit nature of name resolution and function dispatch in Python, accurately tracking the usage and meaning of identifiers is inherently difficult, especially for automated tools. Even widely adopted IDEs like PyCharm may fail to capture all contextual interactions, leading to incorrect or incomplete transformations.

Previous work has investigated the correctness of refactoring tools in statically typed languages such as Java and C by detecting compilation errors, behavioral deviations, and overly strong preconditions [3, 9, 15, 34, 36]. However, the impact of such refactorings on Python, particularly with respect to type errors, remains largely unexplored. To address this gap, there is a clear need for techniques capable of statically analyzing refactoring implementations to identify missing preconditions and prevent runtime type errors. For instance, a robust refactoring tool should verify that renaming a function like `__lt__` to `compare` violates Python's comparison protocol before applying such a transformation. In the following section, we present a technique designed to address this issue by statically detecting type errors introduced during refactoring in Python.

3 Detecting Type Errors in Refactorings

Next we present our technique in detail (SAFEREFACORPY).

3.1 Overview

Our technique takes as input a Python program, the refactoring implementation under test, the location where the refactoring should be applied, and any required parameters for the selected refactoring (Step 1). The location parameter specifies the start and end offsets of the region in the source code to which the refactoring will be applied. If the refactoring tool under test throws any exceptions during the transformation process, our technique records this outcome as-is in the generated bug report. Next, we perform type error analysis on both the original and the refactored versions of the program (Step 2). The differential analysis algorithm then receives both versions of the program and their respective type-checking reports (Step 3). This algorithm identifies discrepancies and classifies the resulting failures (Step 4), ultimately assembling a bug report. Figure 1 provides an overview of all steps in our technique.

3.2 Apply Refactoring

In the first step, we apply the refactoring implementation. For instance, consider the Rename Method refactoring presented in Section 2. For this refactoring type, our technique receives Listing 1 as the input program, the location of the function, and the new name

¹https://docs.python.org/3/reference/datamodel.html#object.__init__

²<https://docs.python.org/3/tutorial/interpreter.html#invoking-the-interpreter>

³https://docs.python.org/3/reference/datamodel.html#object.__lt__

⁴<https://github.com/python-rope/rope/issues/773>

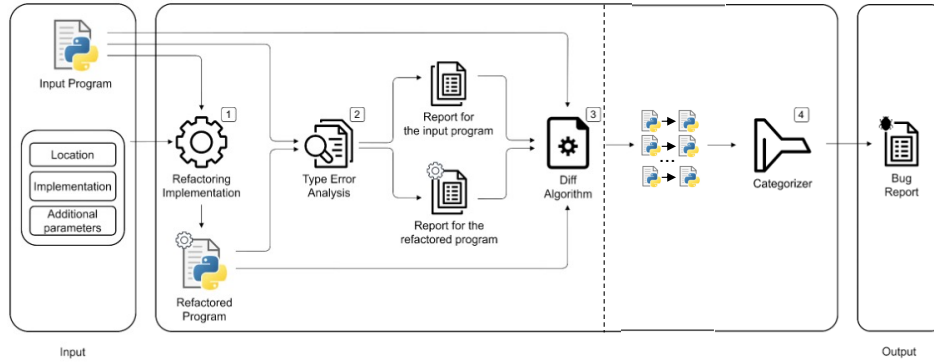


Figure 1: A technique for detecting type errors in Python refactoring implementations.

```

1: main.py:5:4 Missing return annotation [3]: Returning `None` but no return type is specified.
2: main.py:5:23 Missing parameter annotation [2]: Parameter `marks` has no type specified.
3: main.py:5:30 Missing parameter annotation [2]: Parameter `fp` has no type specified.
4: main.py:6:8 Missing attribute annotation [4]: Attribute `_marks` of class `Mark` has no type specified.
5: main.py:11:4 Missing return annotation [3]: Return type is not specified.
6: main.py:14:4 Missing return annotation [3]: Return type is not specified.
7: main.py:14:21 Missing parameter annotation [2]: Parameter `other` has no type specified.
  
```

(a) Type error report of the input program.

```

1: main.py:5:4 Missing return annotation [3]: Returning `None` but no return type is specified.
2: main.py:5:23 Missing parameter annotation [2]: Parameter `marks` has no type specified.
3: main.py:5:30 Missing parameter annotation [2]: Parameter `fp` has no type specified.
4: main.py:6:8 Missing attribute annotation [4]: Attribute `_marks` of class `Mark` has no type specified.
5: main.py:11:4 Missing return annotation [3]: Return type is not specified.
6: main.py:14:4 Missing return annotation [3]: Return type is not specified.
7: main.py:14:22 Missing parameter annotation [2]: Parameter `other` has no type specified.
8: main.py:22:6 Unsupported operand [58]: `<` is not supported for operand types `Mark` and `Mark`.
  
```

(b) Type error report of the refactored program.

```

Missing return annotation [3]
Missing parameter annotation [2]
Missing parameter annotation [2]
Missing attribute annotation [4]
Missing return annotation [3]
Missing return annotation [3]
Missing parameter annotation [2]
  
```

(c) Type errors of the input program.

```

Missing return annotation [3]
Missing parameter annotation [2]
Missing parameter annotation [2]
Missing attribute annotation [4]
Missing return annotation [3]
Missing return annotation [3]
Missing parameter annotation [2]
Unsupported operand [58]
  
```

(d) Type errors of the refactored program.

```

Unsupported operand [58]
  
```

(e) Resulting set.

Figure 2: Type error analysis of the input and refactored programs.

to apply the refactoring (for example, compare). The parameters consider the location as an element of the parameters to apply the refactoring instance. In this example, we use the Rope refactoring implementation. Our algorithm identifies the location of the function `__lt__` using Python's Abstract Syntax Tree (AST) library. Finally, it yields the refactored program. For other types of refactoring, we apply a similar strategy with adjusted parameters as needed.

3.3 Type Error Analysis

The technique executes the type error analysis on the input and refactored program versions (Step 2). We instantiate our technique using Pyre 0.9.18 [14] as our tool to detect type errors. The tool analyzes one program at a time and produces one report for each program. Consider the input program outlined in Listing 1 and the

refactored version. The reports generated for input and refactored programs are presented in Figures 2a and 2b, respectively.

3.4 Differential Algorithm

In Step 3, our technique receives the reports generated from Step 2. In this step, our goal is to identify the new type errors introduced by the refactored implementation in the resulting program. For instance, consider the Pyre 0.9.18 [14] report of the input program in Figure 2a and the report of the refactored program in Figure 2b. The type error can be identified between the file's name and the description of the detected type error. First, our algorithm removes the file name, line number, and all text following the colon, retaining only the core type error message, as illustrated in Figures 2c and 2d. Then, we focus only on the set of new type errors added

after applying the refactoring. We drew inspiration from the Differential Test [12] to identify differences in the reports. The set of new type errors introduced in the refactored program is presented in Figure 2e.

3.5 Categorizer

After applying all transformations from a refactoring implementation under test in Steps 1-3, some of them may introduce new type errors. However, the resulting set of transformations can be large and diverse, making manual analysis both time-consuming and error-prone. Additionally, multiple failures may stem from the same underlying bug. To address these challenges, we developed a categorizer to group similar failures and streamline the analysis process.

We group the failures based on the type errors introduced after each transformation applied by a refactoring instance. These failure groups are then used to categorize the N failures into bug candidates. For example, if N function renamings result in the same type error as shown in Figure 2e, we randomly select one representative from the group for manual analysis and consider it a potential bug candidate. As another example, during our evaluation of the Rename Method refactoring, we identified three transformations that introduced the following type errors: *Unsupported operand*, *Missing global annotation*, and *Incompatible variable type*. Additionally, one transformation resulted in *Call error* and *Invalid class instantiation*. For each group of similar type errors, we selected one representative transformation as a bug candidate.

3.6 Bug report

In the last step, we may have large input and refactored programs to be analyzed in each transformation. Reporting bugs using large code snippets brings a complexity of understanding to developers. So, we simplified the input program drawing inspiration from the delta debugging technique [24, 45] to improve this scenario. We remove some code snippets in the input program, apply the refactoring with the same parameters, and check whether the new refactored program yields the same new type error. Otherwise, we put back the removed code snippet. We repeat this process until we cannot remove any Python construct in the input program anymore. For example, we utilized the code available on the TextBlob repository on GitHub⁵ to reduce it to the example presented in Section 2 which represents the bug⁴ reported to developers.

4 Evaluation

In this section, we evaluate our technique.

4.1 Definition

The goal of this study [2] is to evaluate our technique for the purpose of analyzing the refactoring implementations for Python with respect to its ability to detect type errors from the viewpoint of researchers in the context of transformations applied to open-source projects. We address the following research questions:

RQ₁ To what extent can our technique detect type errors in refactoring implementations? To answer this question, we

count the number of transformations applied by refactoring implementations that introduce type errors.

RQ₂ What are the type errors detected by our technique? To answer this question, we list the distinct type errors found by our technique.

RQ₃ To what extent are the reported bugs accepted by developers? To answer this question, we count the number of bug reports accepted by developers.

4.2 Methodology

4.2.1 Subject. We selected the *TextBlob* project, version 0.17.1, as the subject of our evaluation. TextBlob is a natural-language processing library for Python (compatible with versions 2 and 3), consisting of over 3,000 lines of code. It has also been used in prior studies [5, 42].

4.2.2 Refactoring Implementations. We evaluated refactoring types supported by Rope version 1.3.0 [27]: Rename Field, Rename Method, Inline Method, Extract Method, Move Field/Method, and Use Function. These refactorings were selected based on their practical relevance and frequency of use in real-world development scenarios [11, 18]. They also present challenges specific to Python's dynamic typing. Without static type checking, transformations such as renaming or moving methods and attributes may silently break code that depends on dynamic features like reflection, string-based patterns, or duck typing. Refactorings in Python often interact with constructs such as dynamically declared attributes (`__init__`), private methods using PEP 8 naming conventions, and operator overloading (e.g., `__lt__`, `__add__`). Moreover, method extraction and inlining can unintentionally eliminate necessary object context (`self`) or duplicate logic. Name shadowing is also a risk when renamed identifiers conflict with built-ins or global variables.

4.2.3 Tooling and Environment. All experiments were performed on a 2.60GHz six-core machine with 32GB of RAM. We used Rope 1.3.0 to programmatically apply refactorings. Rope requires as input the root directory of the project, the file path containing the target, and the exact character offset of the target element. For type error detection, we used Pyre 0.9.18 [14], a static type checker developed and maintained by Meta.

4.2.4 Procedure. We first parsed the modules of the TextBlob project to extract all method and field names using the Python abstract syntax tree (AST). Then, for each refactoring type, we randomly selected a target name from the corresponding set of elements. Each execution of our technique involved a specific refactoring implementation, a selected location in the code, and the original source program. After applying the transformation, we analyzed both the original and the refactored versions using Pyre to detect newly introduced type errors. In our evaluation, we consider that a correct transformation should not introduce any new type errors. This automated setup allowed us to execute and analyze hundreds of transformations, which would not be feasible with a manually curated evaluation and increases the likelihood of uncovering bugs in refactoring implementations.

4.3 Results

⁵<https://bit.ly/textBlobFile>

RQ₁. Our technique applied a total of 1,152 refactoring instances, identifying 480 failures in four refactoring types. Among these, 354 transformations were correctly applied without introducing any type errors, while 197 instances could not be executed due to limitations in the tools or internal crashes. In some cases, the tools themselves reported that the transformation could not be applied, such as when inlining functions with multiple return statements. Table 1 summarizes the number of available targets (*Variables*), failures introducing new type errors, correct applications, and false positives, across different strategies and refactoring types. Strategies such as *Rename Field – Method names* and *Rename Method – Field names* demonstrated higher success rates, achieving 132 and 104 correct applications, respectively. Conversely, the *Rename Field – Keywords* strategy exhibited the highest failure rate, with 166 failures out of 167 attempts. The *Cannot Apply* column highlights cases in which the refactoring tools failed to produce output, typically due to tool crashes or unhandled edge cases.

| Refactoring | Strategy | Variables (targets) | Cannot Apply | Failures | False Positives | Correct application |
|-------------------|--------------------|---------------------|--------------|----------|-----------------|---------------------|
| Inline Method | Method names | 150 | 50 | 21 | 15 | 64 |
| Rename Field | Keywords | 167 | 1 | 166 | 0 | 0 |
| Rename Field | Method names | 167 | 1 | 30 | 4 | 132 |
| Rename Method | Keywords | 150 | 0 | 150 | 0 | 0 |
| Rename Method | Field names | 150 | 1 | 39 | 6 | 104 |
| Use Function | Method names | 28 | 6 | 0 | 0 | 22 |
| Extract Method | Method names | 166 | 0 | 62 | 103 | 1 |
| Move Method/Field | Method/Field names | 174 | 138 | 12 | 23 | 1 |

Table 1: Summary of the automated application of refactoring implementations and selected strategies. Strategy = Input types used as a parameter; Variables (targets) = the number of available targets to apply the selected refactoring type; Cannot Apply = the refactoring cannot be applied; Failures = number of refactoring applications that yield new type errors; False Positives = number of false positives; Correct Application = number of refactoring instances that do not introduce type errors.

RQ₂. Our technique identified 18 distinct type errors. Table 2 presents a summary of these errors, grouped by refactoring type and strategy used. The results reveal that certain type errors are recurrent across different refactoring implementations. The detected errors span several categories, including: (i) *compilation issues* such as *Parsing failure* and *Unexpected keyword*; (ii) *scope-related errors* like *Unbound name* and *Undefined attribute*; (iii) *structural problems* such as *Call error* and *Unsupported operand*; and (iv) *type violations* including *Inconsistent override* and *Invalid class instantiation*. These results demonstrate the diversity and depth of errors that our technique is capable of uncovering across multiple scenarios.

RQ₃. After applying Step 4, we classified 480 failures into 27 distinct bug reports. Table 3 provides a detailed overview of each reported bug. It includes both issues accepted by the JetBrains PyCharm team (IDs prefixed with 'PY') and those submitted to the Rope repository on GitHub. In one case (ID 10), we reported an issue related to a specific refactoring type, but the fix was applied to two refactoring types; thus, we did not file a separate report. Additionally, Table 3 also includes manually identified bugs inspired by those automatically detected by our technique: one in JetBrains (ID 3) and another in the Rename Class refactoring (ID 29). These examples suggest that discovering a bug in one refactoring implementation can help

developers reason about and identify missing preconditions in other implementations.

4.4 Discussion

Next we discuss our results.

4.4.1 Parameters. Refactoring implementations typically require specific parameters to apply transformations correctly. As illustrated in Section 2, we demonstrated the application of the Rename Method refactoring to the program shown in Listing 1. In this scenario, two key parameters must be provided: the location of the function to be renamed and the new name to assign. Based on the program in Listing 1, our technique identifies all functions in the code and applies the refactoring to a selected subset of them. Similarly, other refactoring types evaluated in our study require analogous parameters, such as the location of a field, a new identifier name, or a function body to be extracted or inlined.

To guide the refactoring transformations in our technique, we employ three name selection strategies: “*Project Method Names*”, “*Project Field Names*”, and “*Keywords*”. These strategies are inspired by prior work [30, 31, 34], which used similar mechanisms to reveal bugs in refactoring implementations. The “*Project Method Names*” and “*Project Field Names*” strategies extract existing function and field names from the target project, while the “*Keywords*” strategy uses reserved keywords from the Python language as input values. For example, the bug presented in Section 2 was uncovered using the “*Project Method Names*” strategy.

These strategies are particularly relevant in Python, where the language permits functions and fields to share names across global and local scopes. This flexibility introduces ambiguity, increasing the complexity of correctly applying refactorings – particularly when deciding whether or not a transformation should be performed in a given context. Furthermore, using Python keywords as candidate names allows deeper exploration of corner cases and potential semantic conflicts within refactoring implementations.

Applying these strategies, we identified a total of 29 bugs. The “*Keywords*” strategy uncovered 1 bug in the Rename Field and Rename Method refactorings (both handled by a single Rope implementation). In addition, the “*Project Field Names*” strategy led to 12 bugs (affecting Rename Method and Rename Class), while the “*Project Method Names*” strategy revealed 18 bugs across Rename Field, Rename Method, Inline Method, and Rename Class refactorings. As future work, we plan to extend these strategies to incorporate additional naming patterns, such as special characters or combinations of alphanumeric and symbolic characters, to further challenge the robustness of refactoring implementations.

4.4.2 Type Errors. Our technique using different strategies detected the *Undefined attribute*, *Incompatible parameter type* and *Unsupported operand* type errors in 3 refactoring types (Inline Method, Rename Field, and Rename Method). Pyre [14] emits *Undefined attribute* when an attribute (like a function or a property) is accessed on an object, and Pyre cannot find this attribute in the class definition or inferred type of the object. For example, in Listing 2 the function `show_age` tries to print the `age` attribute of a `Dog` object. However, the `age` attribute is not defined anywhere in the `Dog`

| Refactoring | Inline Method | Rename Field | Rename Field | Rename Method | Rename Method |
|-------------|-----------------------------|------------------------------|-----------------------------|------------------------------|--|
| Strategy | Method names | Keywords | Method names | Keywords | Field names |
| Added Types | Undefined attribute | Undefined attribute | Undefined attribute | Undefined attribute | Undefined attribute |
| | Incompatible parameter type | Incompatible parameter type | Incompatible parameter type | Incompatible parameter type | Incompatible parameter type |
| | Unsupported operand | Unsupported operand | Unsupported operand | Unsupported operand | Unsupported operand |
| | - | Missing global annotation | Missing global annotation | Missing global annotation | Missing global annotation |
| | Call error | Call error | Call error | - | Call error |
| | Uninitialized local | - | Uninitialized local | - | Uninitialized local |
| | Unbound name | - | Unbound name | - | - |
| | Parsing failure | Parsing failure | - | Parsing failure | - |
| | Unexpected keyword | - | - | - | Unexpected keyword |
| | Invalid class instantiation | - | - | - | Invalid class instantiation |
| | Unable to unpack | - | - | - | Unable to unpack |
| | - | - | Incompatible variable type | - | Incompatible variable type |
| | - | Invalid decoration | - | Invalid decoration | - |
| | - | Missing attribute annotation | - | Missing attribute annotation | - |
| | - | Too many arguments | - | Too many arguments | - |
| | - | Undefined or invalid type | - | Undefined or invalid type | - |
| | - | - | - | - | Missing annotation for captured variable |
| | - | - | Inconsistent override | - | Inconsistent override |
| Total | 10 | 10 | 9 | 9 | 12 |

Table 2: Summary of distinct detected type errors per refactoring type applied with Rope that emitted new type errors. Strategy = parameters used by refactoring; Added Types = new type errors introduced by refactoring implementations.

| ID | Refactoring | Issue | Issue trackers IDs |
|----|---------------|--|--------------------|
| 1 | Inline Method | Inline Method refactoring is allowed in methods of the descriptor protocol | 742 |
| 2 | Inline Method | Inline method refactoring inserts an unexpected argument | 757 |
| 3 | Inline Method | Cannot inline functions with the same name as different functions from another package used in the module | PY-66251 |
| | Rename Method | Refactoring custom methods touches library methods of the same name | |
| 4 | Inline Method | Applying the Inline Method refactoring does not add the required import | 743 |
| 5 | Inline Method | Inline Method refactoring is allowed in abstract methods | 744 |
| 6 | Inline Method | Compilation error after applying the Inline method refactoring | 745 |
| 7 | Inline Method | Inline method refactoring applied to rich comparison methods | 758 |
| 8 | Inline Method | Inline method refactoring changes variables names after applying the transformation | 759 |
| 9 | Inline Method | Inline method refactoring passes the wrong parameter to the inlined function body | 760 |
| 10 | Rename Method | Rename refactoring allow the use of Python keywords | 698 |
| | Rename Field | | |
| 11 | Rename Field | Rename Field refactoring allows you to rename a field with the same name used in a global method | 761 |
| 12 | Rename Field | Rename Field refactoring allows you to use the name of special methods as a new name | 762 |
| 13 | Rename Field | Rename refactoring doesn't rename a function's default arguments when the renamed variable is defined in the class scope | 686 |
| 14 | Rename Field | Rename Field refactoring allows the use of declared method names as new field names | 763 |
| 15 | Rename Field | Rename Field refactoring allows the use of a name that can not be iterable | 764 |
| 16 | Rename Field | Rename Field refactoring allows you to rename a class field with class method names | 765 |
| 17 | Rename Field | Rename Field refactoring allows you to rename a local field with the method name | 766 |
| 18 | Rename Field | Rename Field refactoring allows you to change the method parameter, causing inconsistent override | 767 |
| 19 | Rename Method | Rename Method refactoring allows the use of previously declared field name | 768 |
| 20 | Rename Method | Rename Method refactoring allows you to rename a method to a name with an 'internal use' indicator | 769 |
| 21 | Rename Method | Rename Method refactoring does not change the name of the super method in the classes that override it | 770 |
| 22 | Rename Method | Rename Method refactoring is allowed in methods of the descriptor protocol | 771 |
| 23 | Rename Method | Rename Method refactoring does not change the calls of the renamed method | 772 |
| 24 | Rename Method | Rename Method refactoring allowed for methods designed for numeric type emulation | 773 |
| 25 | Rename Method | Rename Method refactoring does not rename all implementations of an abstract method | 774 |
| 26 | Rename Method | Rename Method refactoring allowed for methods defined to implement container objects | 775 |
| 27 | Rename Method | Rename Method refactoring allows you to rename a nested method with a parameter name | 776 |
| 28 | Rename Method | The Rename Method refactoring is allowed in the overridden method | 746 |
| 29 | Rename Class | Rename refactoring doesn't apply to the references of the renamed class | 700 |

Table 3: Summary of the bugs reported to Rope project. ID = identifier; Refactoring = refactoring type; Issue = short description of the reported bug; Issue trackers IDs = the ID of the reported issue in the project's repository (numbers starting with 'PY' represent JetBrains issue tracker (Link: <https://youtrack.jetbrains.com/issue/PY-<ID>>); otherwise, the GitHub (Link: <https://github.com/python-rope/rope/issues/<ID>>) repository).

class, leading to the *Undefined Attribute* type error when checked by Pyre.

The type error *Incompatible parameter type* is thrown when an argument into a function call does not match the expected parameter type of that function. For instance, send a string to a function that expects an integer. The type error *Unsupported operand* refers to operators not supported – for example, $a < b$ when a is a class that

not accept this comparison type. Both type errors are detected by our technique during the application of the Inline Method, Rename Field, and Rename Method refactoring types.

```
class Dog:
    def show_age(self):
        print(self.age) # 'Undefined Attribute'
```


Listing 2: An example demonstrating an *Undefined Attribute* type error within the class definition itself.

```
class Cat      # 'Parsing Failure'
    pass
```

Listing 3: An example within a class construct demonstrating a *Parsing Failure* type error.

Another type errors detected are the *Parsing failure* and *Unexpected keyword* type errors. These type errors occur when the source code does not comply with Python's conventions. For example, in Listing 3 the error is due to the missing colon (:) at the end of the class declaration line (class Cat). This is a syntax error, as Python expects a colon at the end of a class definition line. This kind of mistake would prevent Pyre from parsing the class correctly, leading to the *Parsing Failure* error. Our technique identified 3 refactoring types (Inline Method, Rename Field, and Rename Method) introducing those errors.

A list of distinct type errors identified after executing the technique for various refactoring implementations may be found in Table 2. For example, using the function names strategy when evaluating the Inline Method refactoring, our technique identified 10 distinct type errors introduced.

4.4.3 Test Input Programs. In the first step of our approach, a Python program is provided as input. The technique is versatile and can be applied to a variety of Python projects. For our evaluation, we selected the open-source project TextBlob, which has also been used in prior studies [5, 42]. TextBlob includes 77% of Python's language keywords, offering substantial coverage that enabled the detection of multiple refactoring-related bugs. However, it does not include certain keywords such as `async`, `await`, `del`, `with`, `nonlocal`, `global`, `finally`, and `yield`. These omissions may limit our ability to detect type errors associated with language constructs that rely on these keywords. As future work, we plan to broaden our evaluation by incorporating a more diverse set of Python programs that exercise a wider range of language features.

For instance, consider the code snippet provided in Listing 4. Based on the parameter value, this code may conditionally create an async task. If the parameter is set to false, the async task is not created, and attempting to use the `await` command in such a scenario would result in a type error. In this example, Pyre would emit the *Incompatible Awaitable Type* type error, and our technique would detect those type errors when they are introduced in the refactored program.

```
from asyncio import create_task, sleep
async def create_new_task(flag):
    if flag:
        task = create_task(sleep(1))
    else:
        task = None
    await task
```

Listing 4: An example demonstrating an *Incompatible Awaitable Type* type error.

The code presented in Listing 5 contains an async generator⁶ that allows asynchronous iteration. It is defined using `async def`

and contains `yield` statements. Async generators are ideal when you need to produce values over time while still maintaining the responsiveness of your application. In this example, Pyre would emit the *Incompatible Async Generator Return Type* type error, and our technique would detect those type errors when they are introduced in the refactored program.

```
from typing import AsyncGenerator
async def f() -> int:
    yield 0
async def g() -> AsyncGenerator[int, None]:
    if False:
        yield 1
```

Listing 5: An example demonstrating an *Incompatible Async Generator Return Type* type error.

4.4.4 Bugs. We used a combination of GitHub labels, developer comments, and issue resolution status to assess bug acceptance. While GitHub automatically assigns the "bug" label when an issue is reported, Rope maintainers can review and update the label during triage as needed. If the label remains after review, we interpret it as implicit acceptance. In some cases, developers changed the bug tag to enhancement. Explicit rejections typically include the invalid label. Issues closed with an associated fix were considered both accepted and resolved.

The transformation (ID 24) shown in Section 2 presents a bug⁴ reported to developers of Rope project during application of Rename Method refactoring. Our technique identified a failure category with one type error: the *Unsupported operand* type error. Pyre emits the type error due to the default value in the argument not being renamed after applying the refactoring implementation. Rope developers accepted and indicated they would not correct one of the reported bugs. They understood that the reported bug ID 29 (see Table 3) should be the user's responsibility to ensure that the target name made to apply the Rename Class. Moreover, the developers indicated that they may add a function in Rope's front-end interface to check whether the destination name exists and is available.

In some cases, the refactoring implementation crashes. For example, consider Listing 6 as an input program to apply the Inline Method refactoring to the function `get_string`. The refactoring implementation of Rope crashes. Our technique found a number of crashes similar to this one.

```
def bar():
    s = get_string(1)
    print(s)
def get_string(num):
    if num == 1:
        return 'hello'
    return ''
```

Listing 6: A crash in Rope caused by the Inline Method.

Some bugs (e.g., IDs 22 and 24) are specifically related to Python's dynamic typing and would not typically occur in statically typed languages. These include issues involving Python-specific constructs such as abstract functions, modules, attribute declarations, and private methods that follow PEP naming conventions, as well as type emulation behaviors using operators like `<` and `+`.

4.4.5 False Positives. To ensure that program context is preserved, we analyze complete programs during Steps 1 and 2. In Step 3,

⁶<https://docs.python.org/3/library/typing.html#typing.AsyncGenerator>

however, we rely on diffs to isolate changes and identify type errors introduced by refactoring. This diff-based strategy helps reduce false positives that might otherwise arise from running Pyre on the entire refactored program. Nonetheless, it also introduces the risk of false negatives—particularly when subtle issues are not captured in the diff or when a bug is indirectly related to a change. Additionally, some bugs may be missed or misclassified during the grouping process in Step 4. While Pyre itself may produce false positives, our approach of focusing solely on differences helps narrow the analysis to errors that are more likely to be introduced by the transformation. We recommend using the technique iteratively: developers can address a subset of detected issues and re-run the analysis to refine the results and capture remaining issues.

In some cases, developers did not accept the reported bugs, which may indicate potential false positives or reflect differing interpretations of refactoring correctness. For instance, in one of the reported cases, the Rope developers argued that it is the user's responsibility to avoid naming conflicts. However, in Java refactoring implementations, such conflicts are typically detected and prevented by the tool itself, as they can cause compilation or semantic errors. From our perspective, similar preconditions should also be enforced by Python refactoring tools, and such cases should be treated as bugs.

4.4.6 Pytype. Pyre may face some challenges related to false positive and false negatives. An alternative tool for identifying type errors in refactored programs is Pytype⁷, a static type analyzer developed by Google. Pytype analyzes Python code to check and infer types without requiring type annotations, flags common type-related mistakes, supports linting, and can optionally enforce user-provided type annotations or generate them in standalone files. Unlike Pyre, which may exhibit limitations when analyzing dynamically typed or annotation-free code, Pytype is specifically designed to perform well even in the absence of type hints. Incorporating Pytype into our workflow may help reduce false negatives – i.e., type errors that are not detected by Pyre—thereby enhancing the overall robustness of our type error detection process.

```
def normalizer():
    return ''
class Word:
    word = normalizer()
    def singular(self, parser=word):
        return parser.singular('')
```

Listing 7: Input program to apply the Inline Method.

Some type errors detected by Pytype and Pyre are similar, as both tools are designed to identify type-related issues in Python code. However, their internal error classification schemes and naming conventions may differ. For instance, when applying our technique to the transformation from Listing 7 to Listing 8, Pytype reports a *name-error*, which corresponds to a type error that Pyre also detects. This demonstrates that our technique is capable of identifying type issues consistently across different type analysis tools. As future work, we plan to integrate Pytype into our technique by adapting Step 3 (Section 3.4) to evaluate its effectiveness in uncovering additional bugs that may be missed by Pyre, potentially increasing the coverage and reliability of our analysis.

⁷<https://google.github.io/pytype/>

```
def normalizer():
    return ''
class Word:
    word = normalizer()
```

Listing 8: Program after apply the Inline Method refactoring.

4.4.7 Other Tools. As a feasibility study, we verified whether the same bugs occur in IDEs like PyCharm⁸ Community 2023.1, PyDev⁹ 10.1.4, and VSCode¹⁰ 1.81.0. For each bug, we used the same initial code and parameters described in the corresponding bug report, manually applied the refactoring in the IDE, and analyzed the output using our technique. Our analysis revealed that some of the same bugs are also present in PyCharm (ID 3¹¹ in Table 3) and in PyDev (IDs 10 and 13 in Table 3). These examples suggest that identifying a bug in one refactoring implementation can help developers reason about and uncover missing preconditions in others. As future work, we intend to broaden our evaluation to include additional IDEs and refactoring engines. To enable automated testing in new environments, it is necessary to access the refactoring implementation interfaces and adapt Step 1 of our technique (Section 3.2) accordingly. Notably, the remaining steps of the technique do not require modification, demonstrating its adaptability and potential for integration with diverse refactoring tools.

4.4.8 Implementation Effort. Although our technique is general and extensible, working with Rope posed practical challenges due to its limited documentation, which assumes prior familiarity with its internal architecture. This resulted in a steep learning curve and required engineering effort. Programmatic use of Rope demands a detailed understanding of its API. For instance, applying the Rename Field refactoring to an attribute located on the third line of a file requires computing the exact character offset, accounting for all preceding characters and line breaks. Familiarity with the refactoring implementations may help developers integrate our technique more easily.

4.4.9 Unit Tests. Unit tests can detect certain refactoring issues, particularly those that result in observable behavioral changes. However, their effectiveness is limited by the availability and quality of test suites, as well as the developer's ability to identify which parts of the system are affected by the refactoring [25]. Our static analysis technique complements unit testing by automatically analyzing the static type environment before/after refactoring operations. By leveraging tools such as Pyre, it can detect type-related violations introduced by transformations, even in the absence of test cases. While effective at identifying type errors, our approach doesn't capture all semantic behavior changes. Ideally, it would be integrated with dynamic techniques that automatically generate tests for the refactored entities to verify behavioral preservation [16, 35]. We plan to address this in future work.

4.5 Threats to Validity

One potential threat lies in the manual analysis of refactoring outcomes. Step 4 was manually performed by the first author. After the

⁸ <https://www.jetbrains.com/pt-br/pycharm/>

⁹ <https://www.pydev.org/>

¹⁰ <https://code.visualstudio.com/docs/python/python-tutorial>

¹¹ <https://youtrack.jetbrains.com/issue/PY-66251>

initial bug reports were created, the second author independently reviewed them. In a few cases—particularly those involving subtle aspects of Python’s semantics—there were disagreements, which were resolved through discussion with a third author. This process is inherently error-prone and may be influenced by individual bias or misinterpretation. However, all identified bugs discussed were submitted to tool maintainers, and some of them were confirmed and accepted, which lends credibility to the analysis.

Another threat relates to the accuracy of the type error detection process. Our technique relies on Pyre to identify type errors through static analysis. Although Pyre is a widely used and robust tool, it may produce false positives or false negatives, particularly due to Python’s dynamic typing and Pyre’s limitations in handling complex typing scenarios. To extract the type error locations, our technique processes Pyre’s verbose textual output, which may also introduce parsing inaccuracies. Despite these limitations, Pyre provides a practical and consistent basis for evaluating type safety in Python, and our analysis strategy was carefully designed to minimize false positives. However, further evaluation is necessary.

Our evaluation was limited to a subset of popular refactoring types, including Rename Field, Rename Method, Inline Method, Extract Method, Move Field/Method, and Use Function. While these are widely used in practice [18], our findings may not extend to more specialized or complex refactorings. Future work will expand the scope to cover additional refactoring types and scenarios.

We evaluated refactoring implementations primarily from the Rope library. Although Rope is commonly used in the Python ecosystem, evaluating implementations from additional tools and IDEs such as PyCharm and VSCode would strengthen the generalizability of our conclusions. Some of the bugs identified in Rope were also observed in PyCharm and PyDev. As discussed in Section 4.4.7, our technique is designed to be adaptable to other refactoring engines and type-checking tools.

Although we used real-world open-source projects, including TextBlob, and selected projects with at least 70% Python keyword coverage (a metric used in prior studies [5, 42]), these projects may not fully represent the diversity of Python programs in the wild. Different codebases may exercise other language features or rely more heavily on dynamic constructs, which could impact the effectiveness of our technique.

5 Related Work

Opdyke and Johnson [21, 22] coined the refactoring term, describing the process and identifying common refactorings. Roberts [26] automated the basic refactorings proposed by Opdyke. Later, Tokuda and Batory [41] demonstrated that the preconditions proposed by Opdyke are not sufficient to guarantee behavior preservation after applying transformations. Moreover, proving refactorings concerning formal semantics considering all language constructs constitutes a challenge [31]. AlOmar et al. [1] performed a systematic mapping study on behavior preservation during software refactoring, providing a comprehensive overview of current practices, challenges, and research gaps in the field.

Daniel et al. [3] proposed an approach for automatically testing refactoring implementations in Java. Their technique is built

on ASTGEN, a Java program generator, and relies on a set of programmatic oracles to assess the correctness of refactorings. They developed six oracles to evaluate the outputs of refactoring transformations and applied the technique to 42 refactoring implementations. Building on this work, Gligoric et al. [10] introduced UDITA, a Java-like language that extends ASTGEN by supporting a hybrid approach to test input generation. Unlike ASTGEN, which follows a purely generative style, UDITA allows developers to express constraints using a combination of filtering and generation strategies, making it both more expressive and efficient in producing relevant test programs. In contrast, our work focuses on testing refactoring implementations in Python rather than Java. Instead of generating synthetic test inputs, we leverage real-world open-source Python projects as input programs – similar to the methodology adopted by Gligoric et al. [9] in their empirical testing of refactoring implementations for Java and C.

Steimann and Thies [37] found that mainstream Java IDEs – such as Eclipse, NetBeans, and IntelliJ – exhibit flaws in preserving accessibility during refactorings. They identified scenarios in which the application of common refactorings, such as Pull Up Members, leads to unintended changes in program behavior, highlighting the challenges of ensuring semantic correctness in automated transformations. Soares et al. [34] proposed a technique for detecting behavioral changes and compilation errors introduced by refactoring implementations in Java. By applying their approach to 29 refactoring implementations using automatically generated input programs, they identified 57 bugs related to compilation errors and 63 related to behavioral changes. Similarly, Mongiovi et al. [15] introduced a technique to detect bugs caused by overly strong preconditions in Java refactoring engines. Their method involves generating small Java programs and injecting them into refactoring implementations. They also systematically disable portions of the refactoring logic to identify which preconditions prevent the transformation from being applied, classifying them as potentially overly strong preconditions. In our work, we identified some bugs related to type errors in Python refactoring implementations by applying transformations to real open-source projects, rather than relying on the generation of small synthetic programs.

Tempero et al. [39] conducted a large-scale survey involving 3,785 developers to investigate the barriers to applying refactorings in practice. Their findings indicate that refactoring decisions are often influenced by non-design considerations, and one of the key reasons cited for avoiding refactoring is inadequate tool support. In contrast, our work aims to strengthen the reliability of refactoring tool implementations by statically detecting type errors introduced during transformations. By improving tool robustness, our technique may contribute to increasing developers’ confidence in using automated refactoring tools.

Schäfer [29] discussed the issues and challenges associated with developing refactoring tools for dynamically typed languages. He highlighted the complexity of specifying and verifying preconditions in such environments, as well as the difficulty in determining whether access to module members is safe during transformations. In this context, our technique can support tool developers by statically identifying type errors introduced by refactoring implementations in Python, thereby helping to address some of the challenges outlined by Schäfer.

Wang et al. [43] conducted a comprehensive manual analysis of 518 bugs from three widely used refactoring engines in Java – Eclipse, IntelliJ IDEA, and NetBeans – identifying common root causes, bug symptoms, and characteristics of input programs that trigger faults. Their study yielded a set of actionable findings, which were used to derive guidelines for improving the detection and debugging of refactoring-related defects. Furthermore, their transferability analysis revealed 130 previously unknown bugs in the latest versions of these tools, underscoring the widespread and persistent nature of bugs in refactoring implementations. In contrast, our work evaluates a static analysis technique for detecting type errors introduced by refactoring implementations in Python, using real-world open-source programs as input. While Wang et al. [43] focus on a broad characterization of refactoring bugs in statically typed languages and IDEs, our approach targets type-related issues specific to dynamically typed environments.

Dong et al. [6] proposed a ChatGPT-based approach for testing refactoring engines, leveraging LLMs to automatically generate test programs aimed at uncovering defects. Their method constructs a feature library derived from existing bug reports and test cases, defines preconditions for each refactoring type, and employs predefined prompt templates to guide the generation process. The generated programs are then used in differential testing across multiple refactoring engines, with the results manually analyzed to identify defects. By evaluating seven refactoring types, the authors identified 115 bugs that led to compilation errors or behavioral changes in Java refactoring implementations. In contrast, our approach introduces an automated static analysis technique to test refactoring implementations for Python using real-world open-source projects. Rather than generating synthetic programs through LLMs – an approach that may be constrained by limited context windows – we directly apply refactorings to existing Python codebases, enabling a broader evaluation of tool behavior in practical settings.

Gheyi et al. [8] evaluated the effectiveness of Small Language Models (SLMs) in detecting two categories of refactoring bugs: transformations that introduce compilation or behavioral errors (Type I), and transformations that are incorrectly blocked by IDEs despite being valid (Type II). They evaluated eight language models, including Phi-4 14B and o3-mini-high, using zero-shot prompting to analyze 100 refactoring bugs reported in Java and Python, collected from widely used IDEs such as Eclipse and NetBeans. They highlighted the low computational cost of SLMs, their ability to generalize across languages and refactoring types, and promising results for identifying Type I bugs. However, the models struggled to detect Type II bugs with up to 61 lines of code and often failed to provide accurate explanations, limiting their reliability in complex scenarios. Our work introduced a static analysis technique for detecting type errors introduced by refactoring implementations applied to Python projects. Unlike previous approach, which primarily focus on transformations applied to small, isolated programs due to context window limitations, our technique is capable of analyzing refactorings in larger, real-world programs. However, unlike previous work, our technique does not detect behavioral changes introduced by refactorings. Through our evaluation, our technique uncovered 29 bugs in four refactoring types.

Dilhara et al. [4] proposed PyCraft, a framework that combines static and dynamic code analysis with large language model (LLM)

capabilities to refine code transformations for Python. PyCraft generates diverse code variations along with corresponding test cases to ensure correctness. To assess its effectiveness, the authors submitted 86 transformed code instances across 44 pull requests to open-source projects, achieving an acceptance rate of 83%. These results underscore the potential of integrating LLMs with traditional analysis techniques to support automated code transformation. In contrast, our work focuses on statically detecting type errors introduced by refactoring implementations. Our technique could be integrated into frameworks like PyCraft to increase confidence that the transformations do not compromise type safety.

6 Conclusions

In this paper, we introduced a static analysis technique to identify type errors introduced by refactoring implementations in Python. By applying 1,152 refactoring transformations to a real-world Python project, our technique uncovered 29 bugs of four distinct refactoring types, as well as 18 unique type errors. We submitted all identified bugs to the respective tool maintainers, and a number of them of them were acknowledged and accepted – demonstrating the practical relevance and real-world applicability of our findings.

These results provide empirical evidence that current Python refactoring implementations – including those integrated into widely used IDEs – can introduce type-related bugs, potentially undermining the safety and reliability of automated code transformations. Our findings highlight the need for more rigorous validation mechanisms to ensure that refactorings preserve type correctness, particularly in dynamically typed languages like Python. By acting as an additional verification layer, our technique supports developers and tool maintainers in strengthening the robustness of refactoring workflows and mitigating the risk of subtle, hard-to-detect bugs. Although our evaluation focused specifically on refactorings, the technique is not limited to this context and can be applied to assess other types of code transformations in Python by verifying whether they introduce new type errors.

Looking ahead, we intend to broaden our evaluation to include additional IDEs and refactoring engines, as well as explore the use of PyType as an alternative to Pyre for type error detection. We also plan to expand our technique to cover a wider range of refactoring types and to automatically detect other categories of refactoring-related bugs, such as behavioral changes and overly strong preconditions [3, 15, 30, 34, 36]. Moreover, we aim to investigate the complementary role of Large Language Models (LLMs) and agent-based systems in validating and even repairing refactorings, paving the way for more intelligent and semantically aware refactoring support in future development environments.

ARTIFACT AVAILABILITY

All study artifacts are available online [20].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by CNPq (403719/2024-0, 310313/2022-8, 404825/2023-0, 443393/2023-0, 312195/2021-4), FAPESQ-PB (268/2025).

REFERENCES

- [1] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2021. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology* 140 (2021).
- [2] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 2002. *The Goal Question Metric Approach*. Wiley, 528–532 pages.
- [3] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *Foundations of Software Engineering*. 185–194.
- [4] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. *Proceedings of the ACM on Software Engineering* 1 (2024).
- [5] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering Repetitive Code Changes in Python ML Systems. In *International Conference on Software Engineering*. 736–748.
- [6] Chunhao Dong, Yanjie Jiang, Yuxia Zhang, Yang Zhang, and Liu Hui. 2025. ChatGPT-Based Test Generation for Refactoring Engines Enhanced by Feature Analysis on Examples. In *International Conference on Software Engineering*. 746–746.
- [7] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley, online.
- [8] Rohit Gheyi, Marcio Ribeiro, and Jonhnanthan Oliveira. 2025. Evaluating the Effectiveness of Small Language Models in Detecting Refactoring Bugs. arXiv:2502.18454 [cs.SE] <https://arxiv.org/abs/2502.18454>
- [9] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. 2013. Systematic Testing of Refactoring Engines on Real Software Projects. In *European Conference on Object-Oriented Programming*. 629–653.
- [10] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *International Conference on Software Engineering*. 225–234.
- [11] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One thousand and one stories: a large-scale survey of software refactoring. In *Foundations of Software Engineering*. 1303–1313.
- [12] William McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.
- [13] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *Transactions on Software Engineering* 30, 2 (2004), 126–139.
- [14] Meta Platforms, Inc. 2025. Pyre - A performant type-checker for Python 3. <https://pyre-check.org/>.
- [15] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. 2018. Detecting Overly Strong Preconditions in Refactoring Engines. *Transactions on Software Engineering* 44, 5 (2018), 429–452.
- [16] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. 2014. Making refactoring safer through impact analysis. *Science of Computer Programming* 93 (2014), 39–64.
- [17] Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. 2014. Scaling Testing of Refactoring Engines. In *International Conference on Software Maintenance and Evolution*. 371–380.
- [18] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *Transactions on Software Engineering* 38, 1 (2012), 5–18.
- [19] Jonhnanthan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. 2019. Revisiting the refactoring mechanics. *Information and Software Technology* 110 (2019), 136–138.
- [20] Jonhnanthan Oliveira, Rohit Gheyi, Márcio Ribeiro, and Alessandro Garcia. 2025. Bugs in the Shadows: Static Detection of Faulty Python Refactorings. <https://github.com/jonh-copin/technique>
- [21] William Opdyke. 1992. *Refactoring Object-oriented Frameworks*. Ph. D. Dissertation. UIUC.
- [22] William Opdyke and Ralph Johnson. 1990. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *Symposium Object-Oriented Programming Emphasizing Practical Applications*. 274–282.
- [23] Gustavo Pinto and Fernando Kamei. 2013. What programmers say about refactoring tools? an empirical investigation of stack overflow. In *Workshop on Refactoring Tools*. 33–36.
- [24] Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. 2019. Java reflection API: revealing the dark side of the mirror. In *Foundations of Software Engineering*. 636–646.
- [25] Napol Rachatasumrit and Miryung Kim. 2012. An empirical investigation into the impact of refactoring on regression testing. In *International Conference on Software Maintenance*. 357–366.
- [26] Donald Roberts. 1999. *Practical Analysis for Refactoring*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.
- [27] Rope. 2025. Open source Python refactoring library. <https://github.com/python-rope/rope>.
- [28] G. Van Rossum and F. L. Drake. 2011. *An Introduction to Python*. Network Theory, online.
- [29] Max Schäfer. 2012. Refactoring Tools for Dynamic Languages. In *Workshop on Refactoring Tools*. 59–62.
- [30] Max Schäfer and Oege de Moor. 2010. Specifying and implementing refactorings. In *Object-Oriented Programming Systems Languages and Applications*. 286–301.
- [31] Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2008. Challenge Proposal: Verification of Refactorings. In *Programming Languages Meets Program Verification*. 67–72.
- [32] Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2008. Sound and Extensible Renaming for Java. In *Object-Oriented Programming Systems Languages and Applications*. 277–294.
- [33] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. 2009. Stepping Stones over the Refactoring Rubicon. In *European Conference on Object-Oriented Programming*. 369–393.
- [34] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2013. Automated Behavioral Testing of Refactoring Engines. *Transactions on Software Engineering* 39, 2 (2013), 147–162.
- [35] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. 2010. Making Program Refactoring Safer. *IEEE Software* 27 (2010), 52–57.
- [36] Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. 2011. Identifying overly strong conditions in refactoring implementations. In *International Conference on Software Maintenance*. 173–182.
- [37] Friedrich Steimann and Andreas Thies. 2009. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In *European Conference on Object-Oriented Programming*. 419–443.
- [38] Ke Sun, Yifan Zhao, Dan Hao, and Lu Zhang. 2023. Static Type Recommendation for Python. In *Automated Software Engineering*. 1–13.
- [39] Ewan Tempero, Tony Gorschek, and Lefteris Angelis. 2017. Barriers to Refactoring. *Communications of the ACM* 60, 10 (2017), 54–61.
- [40] Frank Tip, Adam Kiezun, and Dirk Bäumer. 2003. Refactoring for Generalization Using Type Constraints. In *Object-Oriented Programming, Systems, Languages, and Applications*. 13–26.
- [41] Lance Tokuda and Don Batory. 2001. Evolving Object-Oriented Designs with Refactorings. In *Automated Software Engineering*. 89–120.
- [42] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazi-nanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *International Conference on Software Engineering*. 483–494.
- [43] Haibo Wang, Zhuolin Xu, Huaen Zhang, Nikolaos Tsantalis, and Shin Hwei Tan. 2024. An Empirical Study of Refactoring Engine Bugs. arXiv:2409.14610 [cs.SE] <https://arxiv.org/abs/2409.14610>
- [44] Siqi Wang, Xing Hu, Bei Wang, Wenxin Yao, Xin Xia, and Xinyu Wang. 2025. Refactoring Deep Learning Code: A Study of Practices and Unsatisfied Tool Needs. In *International Conference on Software Maintenance and Evolution*.
- [45] Andreas Zeller. 2009. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, online.