Università
della
Svizzera
italiana

**Software
Institute**

PyRef: A Refactoring Detection Tool For Python Projects

Hassan Atwi

June 2021

Supervised by
Prof. Dr. Michele Lanza

Co-Supervised by
Dr. Bin Lin

Software & Data Engineering Master Thesis

Abstract

Refactoring, the process of improving internal code structure of a software system without altering its external behaviors, is widely applied during software development. Understanding how developers refactor source code can help us gain better understandings of the software development process and the relationship between various versions of software systems.

Currently, many refactoring detection tools (*e.g.,* REFACTORINGMINER and REF-FINDER) have been proposed and have received considerable attention. However, most of these tools focus on Java program, and are not able to detect refactorings applied throughout the history of a Python project, although the popularity of Python is rapidly magnifying. Developing a refactoring detection tool for Python projects would fill this gap and help extend the language boundary of the analysis in variant software engineering tasks.

In this work, we present PYREF, a tool that automatically detect 11 different types of refactoring operations in Python projects. Our tool is inspired by REFACTORING MINER, the state-of-the-art refactoring detection tool for Java projects. With that said, while our core algorithms and heuristics largely inherit those of REFACTORING MINER, considerable changes are made due to the different language characteristics between Python and Java.

PYREF is evaluated against oracles collected from various online resources. Meanwhile, we also examine the reliability of PYREF by manually inspecting the detected refactorings from real Python projects. Our results indicate that PYREF can achieve satisfactory precision. Our work not only presents a tool for researchers to conduct refactoring-related studies with Python projects, but also provides the first dataset of refactoring operations in the Python language.

Acknowledgements

This is to acknowledge all those without whom this work would not have been accomplished. First of all, I would like to thank Prof. Dr. Michele Lanza who gave me his immense support and the golden opportunity to do this research work under his supervision. I also want to thank Dr. Bin Lin who dedicated his time towards providing me with invaluable guidance throughout the research. Thanks to all the members of the Software Institute and Dean's Office for your great help and magnificent services.

Last but not least, I am extremely grateful to my parents who always stood by my side and helped me with overcoming numerous obstacles in life with their devoted and wise advice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we first introduce the background information related to refactoring, and then present a brief overview of this thesis. In the end of the chapter, we describe how this thesis is structured.

1.1 Refactoring

1.1.1 History of Refactoring

Maintainability of source code is a key aspect to be accomplished during the software development process, which is essential for developers to have a substantial level of understanding of code logic and structure. To improve the code maintainability and obtain clean code, *refactoring* has been widely applied in practice.

As defined by Martin Fowler, *"refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [14]."* In fact, the first use of the term "refactoring" in the software engineering domain can be traced back to 1990 in a published work by William Opdyke and Ralph Johnson [23]. However, only after the publication of the book "*Refactoring: Improving the Design of Existing Code*" of Martin Fowler in 1999 [8], refactoring gradually got popularized.

Over the past few decades, refactoring has helped developers to address many software design issues and improve the consistency of source code [8]. Moreover, it helps to reinforce the source code's structure to be compatible with future implementations, without altering the key behaviors of the framework.

1.1.2 Emergence of Refactoring Detection Tools

Considering the enormous advantages of refactoring and the popularity of this technique among a vast amount of projects, detecting and studying refactorings in source code can be very rewarding. Practicing reverse engineering methods to extract refactorings has been a huge focus in software engineering research. As a prime example, many empirical studies have been conducted to grasp better understandings of developers' design decisions and logic behind refactorings. Moreover, refactoring detection results allow code reviewers to have deeper insights on the source code they are reviewing, thus facilitating the code review process. On top of that, the learning curve for the changes made in code structure and design can be shortened as the detected refactorings can describe the evolution of the source code throughout the time. Additionally, by correctly detecting refactorings and filtering them out from real bug fixes, we can avoid noise when analyzing the origin of bugs [10]. Automated code completion and refactoring suggestion can also benefit from refactoring detection. However, the detection of refactorings is not a trivial process due to the lack of documentations explicitly recording refactoring operations applied by the developers [20]. Another issue is that most refactorings are performed together with or as a consequence of other changes [21], which makes it even harder to distinguish between refactoring and other code changes.

While there are some difficulties to detect refactoring, given all these benefits of refactoring detection, several refactoring detection tools [29, 18, 26, 12] have been proposed to extract the refactorings in source codes. The process of refactoring detection consists of two main steps, code matching and refactoring identification. With the provided input which consists of the two different versions of a source code, code elements are being matched based on many unique aspects, such as textual similarity, or the complexity of various metrics in case of statement matching for instance. Matching elements between two versions is a crucial step and must be done as accurately as possible, as mismatching two elements can negatively impact the refactoring detection accuracy. With matched code elements, refactoring operations can be identified based on predefined rules.

These refactoring detection tools usually take the same format of input, which consists of two different versions of source code before and after a commit. The output is normally a list of possibly applied refactorings, often including the type and the location of the refactorings. One popular approach to detect refactoring is using fixed similarity threshold values to detect similarity between different entities of the source code. For example, when the tool REFDIFF [26] determines whether two methods can be matched, it checks if the similarity between two methods is above a certain threshold. To obtain the threshold, the authors of the tool applied a calibration process on a randomly selected set of ten commits that contain refactorings from ten different projects. Other approaches, such as REFACTORINGMINER [29], instead adopting heuristics without thresholds to identify refactorings. Due to the fact that code changes are only applied on a specific part of the source code, REFACTORINGMINER [29] focus only on the modified, added or removed elements of the source code and ignore the rest. This was proven to be highly effective, and can identify refactorings in a reliable way [29].

Unfortunately, while many refactoring detection tools have been developed over the last 20 years, there is no such a tool available for Python projects. Most of the existing detection tools concentrate on Java code and its properties. Designing a refactoring detection tool for Python will enable us to extend our knowledge and experience in the field of refactoring detection. In addition, The domains of Python application differ in functionality from those of Java applications. For example, Python's properties such as its dynamic type checking, simple syntax and readability allow fast testing of complex algorithms, which in turn gives data-science related applications a flexible environment where developers can implement their concept directly [24]. Java, instead, is often used in domains such as web services and Android mobile applications. The domain difference can shape the structure of the code and its implementation, hence this can be a source of new issues to be tackled when considering refactorings. Therefore we aim to develop a tool that can mine refactorings in a given Python repository.

1.2 Overview of the Thesis

The main focus of this thesis is to detect refactoring operations from Python projects. Inspired by REFACTORINGMINER [29], we propose a tool named PYREF. PYREF takes as input two versions of a system and detect refactorings only with the changed parts of source code. The tool parses the necessary code elements and then convert them into two ASTs representing the changed code before and after the refactoring. Identical elements on these two ASTs will be matched. After that, with the help of some predefined rules, refactorings can be extracted.

To verify the reliability of our tool, we evaluate the detection reliability of the tool over an oracle consisting of refactoring instances collected from various only resources. We also run our tool on several real Python projects and manually inspect whether the detected refactorings are real ones. Moreover, we also test the running efficiency of PYREF by measuring how long it needs to detect refactorings for projects. Our evaluation results indicate that PYREF can detect refactorings from Python projects efficiently with a high precision. We also discuss the weaknesses of PYREF and point out potential directions for the performance improvement.

1.3 Structure of the Document

This document is structured as follows:

- In Chapter 2, we introduce the various fundamental techniques for refactoring detection and state-of-the-art refactoring detectors. Moreover, we introduce the applications of these tools in software engineering studies. In the end, we sum up the chapter and indicate the limitations of the existing tools.

- In Chapter 3, we present the architecture of our tool PYREF, which is developed to detect refactorings in Python projects. The process of refactoring detection consists of five stages, including: *Extracting Code Changes,* *Refining Code Change Representation, Applying Refactoring Heuristics, Sorting Candidates*, and *Generating Refactoring List*. Each of these stages is thoroughly described in this chapter.

- In Chapter 4, we present two studies to evaluate the performance of PYREF in terms of reliability and efficiency. More specifically, in this chapter we investigate 1) whether PYREF can accurately detect refactorings, and 2) how much time PYREF needs to detect refactoring from a repository.
- In Chapter 5, we recap our work and list the insights we acquired during the development and evaluation of PYREF, followed by the potential future work.

Chapter 2

State of the Art

In this chapter, we first introduce the predecessors in the field of refactoring detection, which lay a solid foundation or have a profound impact for the later refactoring mining tools. We then present the state-of-the-art refactoring detectors and discuss their performance. We also demonstrate the usefulness of these tools by illustrating the applications. In the end, we close this chapter by summarizing the issues of existing refactoring detection tools.

2.1 Fundamental Techniques

Code analysis technologies, especially those that enable code comparison and clone detection, are the main predecessors of the concrete refactoring detection tools.

For instance, CCFINDER (Code Clone Finder) [16] is a clone detection tool developed by Kamiya *et al.*, which allows users to detect a portion of code that is identical or similar to another. CCFINDER transforms input source code to a bag of tokens that is used for further comparison to identify the similarity. CCFINDER was tested on the source code of JDK, FreeBSD, NetBSD, Linux, and many other systems and the result indicate that the tool was highly effective in identifying code clones.

CHANGEDISTILLER [13], built on top of the EVOLIZER [15] platform, is a tool which focuses on mining Java software archives to extract code changes. It can detect more than 40 types of code changes in different code constructs such as method declaration and method body. To extract changes, CHANGEDISTILLER analyzes the ASTs of two subsequent versions of a particular class and represents the code changes with a set of basic tree edit operations, such as insert, delete, move, and update. The authors applied CHANGEDISTILLER on multiple software systems to investigate the co-evolution between comments and source code. Their results indicate that source code and comments are co-changed in the same revision for 90% of the time.

Another significant approach, introduced by Van Rysselberghe and Demeyer [31], leveraged heuristics to reconstruct evolution processes of existing software systems. This approach has been used to detect duplication in large amounts of data, and has contributed to the assessment of some refactoring detection patterns later on.

Inspired by vector space information retrieval, Antoniol *et al.* [9] proposed an automated approach to identify class evolution discontinuities and achieved good performance. Their approach can also be used to identify possible refactoring patterns by calculating the cosine similarity between classes that was encoded into a vector representation.

One popular approach adopted by many refactoring detection tools is UMLDIFF, proposed by Xing and Stroulia [32]. This approach automatically detects structural changes between subsequent versions of object-oriented source code by applying reverse engineering techniques on the logical view of an object-oriented system. The correctness and robustness of UMLDIFF were proven through some real-world case studies.

2.2 Existing Refactoring Detection Tools

Thanks to the above fundamental approaches, the last two decades have seen a rapid growth of refactoring detection tools. Below we introduce the state-of-the-art refactoring detection techniques.

2.2.1 RefactoringCrawler

REFACTORINGCRAWLER, developed by Dig *et al.* [12], is one of these modern tools for refactoring detection. It first uses a fast syntactic analysis based on Shingles encoding, a technique from the information retrieval field, to detect refactoring candidates. After locating the possible refactorings, it then adopts a more expensive semantic analysis to refine the results.

The performance of this tool was tested on three projects: EclipseUI, Struts, and JHotDraw. More specifically, they manually collected the applied refactorings in these projects as the oracle and as a result, their tool achieves a precision between 90% and 100% and a recall between 86% and 100% on this dataset.

2.2.2 JDEvAn

JDEVAN (Java Design Evolution Analysis), a tool developed by Xing and Stroulia [33], detects refactorings by applying a set of predefined queries on the design changes that occurred in two given versions of a system, the composition of change facts is retrieved by implementing the UMLDIFF approach. Moreover, JDEvAn's front-end, represented as a tabbed view, gives users the ability to write their own queries that could extract refactorings based on the UMLDIFF output. The evaluation of this tool was done on two software systems, and all of the documented refactorings were detected.

2.2.3 Ref-Finder

REF-FINDER [18] is another refactoring detection tool developed by Prete *et al.* and based on the tool LSDIFF [19], which is used to identify structural differences. REF-FINDER converts the target source code into logic predicates that describe the structure and elements of the code. After that, it uses Tyruba logic programming engine [11] to infer concrete refactoring instances with the help of template logic queries, which encode the structural constraints of a program before and after refactorings.

It was stated that it could detect around 63 types of complex refactorings throughout the code. The tool was tested on manually collected refactorings and it showed that its overall precision and recall are 79% and 95%, respectively.

2.2.4 RefDiff

Developed by Silva and Valente, REFDIFF [26] detects 13 refactoring types through static analysis and code similarity comparison. As other similar tools, REFDIFF takes as input two subsequent versions of a system, and outputs a list of the found refactorings. The adopted detection algorithm consists of two main phases: Source Code Analysis and Relationship Analysis. Throughout the first phase, two versions of source code are parsed and transformed into high level models that represents the code elements; during the second phase both models are further analyzed to detect the relationships between the identical entities. REFDIFF uses TF-IDF method for the similarity computation of the code elements. The similarity threshold is determined through a calibration process on a set of randomly selected systems. Evaluated with an oracle of refactorings applied by graduate students, this tool achieved an accuracy around 61%.

2.2.5 RefactoringMiner

REFACTORINGMINER, developed by Tsantalis [29], can detect around 55 different types of refactorings. REFACTORINGMINER differs from other similar tools in terms of two aspects: First, it does not require any similarity threshold; second, it is not necessary to provide as input the full snapshot of software. REFACTORINGMINER considers only the changes in the code to achieve high performance and this was proven in a recent study that has demonstrated that only 38% of the change history of software systems can be successfully compiled, and considering the whole system can be a serious limitation and a waste of resources. REFACTORINGMINER implements a light version of UMLDIFF algorithm to match different code elements. Additionally, it adopts two novel prepossessing techniques on the input before starting the matching process, including abstraction and argumentization. These two techniques deal with AST type changes in statements caused by refactoring and sub-expression changes within statements caused by parameterization, respectively. After that, based on some pre-defined detection rules, it detects the possible refactorings in the code. To evaluate REFACTORINGMINER, the authors used a dataset of refactoring instances [25] consisting of 538 commits from 185 GitHub projects, and the final result showed a precision of 97% and a recall of 87%.

2.2.6 Reliability of Refactoring Tools

A study [28] has further investigated the performance of most of the available tools, by adopting a universal benchmark which was adjusted in such a way that every included tool could achieve the best performance. For instance, only the refactoring types that can be detected by all the tools were considered. It proves that REFACTORINGMINER has the best performance among the available tools (a precision of 93% and a recall of 72%), while REFACTORINGCRAWLER has a slightly higher precision of 96% than REFACTORINGMINER, but a lower recall which is 59%. With that said, it is important to note that predicting incorrect refactorings (false positives) might lead to more severe consequences than missing refactoring (false negatives) [29], as it might cause noises during the analysis and impact the decisions based on the empirical study. On the other hand, the study showed that REFACTORINGMINER does not perform well in detecting Move Class and Rename Package refactorings, represented by a very low recall (30% and 11%), the reason of this unsatisfactory performance is due to the lack of necessary information in the input format of REFACTORINGMINER. Since these specific refactorings are based on files classified as not changed, REFACTORINGMINER is not able to detect them.

2.3 Applications of the Refactoring Detection Tools

The refactoring detection tools have been used to serve multiple functionalities and many are beyond just detecting refactorings. Such use cases can be seen in surveying software quality after undergoing modifications or improvements.

A research paper published by K. Stroggylos [27] studied how the code metrics of certain projects are affected when refactorings are applied to the system, to accomplish such study the author had to use refactoring detection tools, such as REFACTORINGCRAWLER, to extract the refactorings. The study indicates that refactoring significantly worsens certain metrics. More specifically, a significant increase in metrics such as LCOM, Ca and RFC indicates sometimes refactoring makes some classes to be incoherent.

Some researchers have investigated the motivation of refactorings [25]. They relied on REFACTORINGMINER to search for refactorings performed in the version history of the selected GitHub repositories and they found that refactoring activity is mainly caused by changes in the requirements rather than motives related to code smell removal.

An empirical study [22] investigates how refactorings can impact the SZZ algorithm, since refactorings are always excluded when applying the SZZ algorithm. The authors used REFDIFF to automatically detect refactorings in code changes and found that 6.5% of lines that are flagged as bug-introducing are actual refactorings,

Another use case of the tools is shown in a research [17] that questions about the practice of type changes. However, to detect the type changes the author had to use REFDIFF as a refactoring detection tool, which extracted 297,543 type changes and their subsequent code adaptations from 129 Java projects containing 416,652 commits. The researchers found that type changes are actually more common than renaming, but the current research and tools for type changes are inadequate.

2.4 Summing Up

To sum up, while several refactoring detection tools have been proposed, they still miss or struggle to implement some important features. For example, some of the tools fail to process nested operations, others are unable to extract certain refactoring patterns. Moreover, some tools were evaluated on a biased oracle. Therefore the reliability of those tools cannot be more carefully inspected. Most importantly, all of the existing tools do not support Python projects.

When developing our tool, we considered the elements that other tools miss, and we tried our best to implement a tool which can extract refactorings from a Python repository in an efficient manner.

Chapter 3

PyRef

3.1 Overview

PYREF is a tool designed to mine refactoring operations in Python projects. The main approach behind PYREF is inspired by REFACTORINGMINER [30], which serves the same purpose as PYREF but only for Java projects. Currently PYREF supports the detection of 11 types of refactoring, including *Rename Method, Rename Parameter, Change Return Type, Extract Method, Inline Method, Move Method, Pull Up Method, Push Down Method, Rename Class, Move Class, and Extract Variable*. Table 3.1 illustrated how these supported refactoring types can take place through some examples.

Our tool PYREF takes as input a repository that must contain Python code and outputs a list of the possible refactoring operations performed in the given repository. In addition, users can specify the commit in which PYREF should detect refactoring.

3.2 Architecture

Fig. 3.1 depicts the architecture of PYREF, which consists of 5 main stages: *Extracting Code Changes, Refining Code Change Representation, Applying Refactoring Heuristics, Sorting Candidates*, and *Generating Refactoring List*.

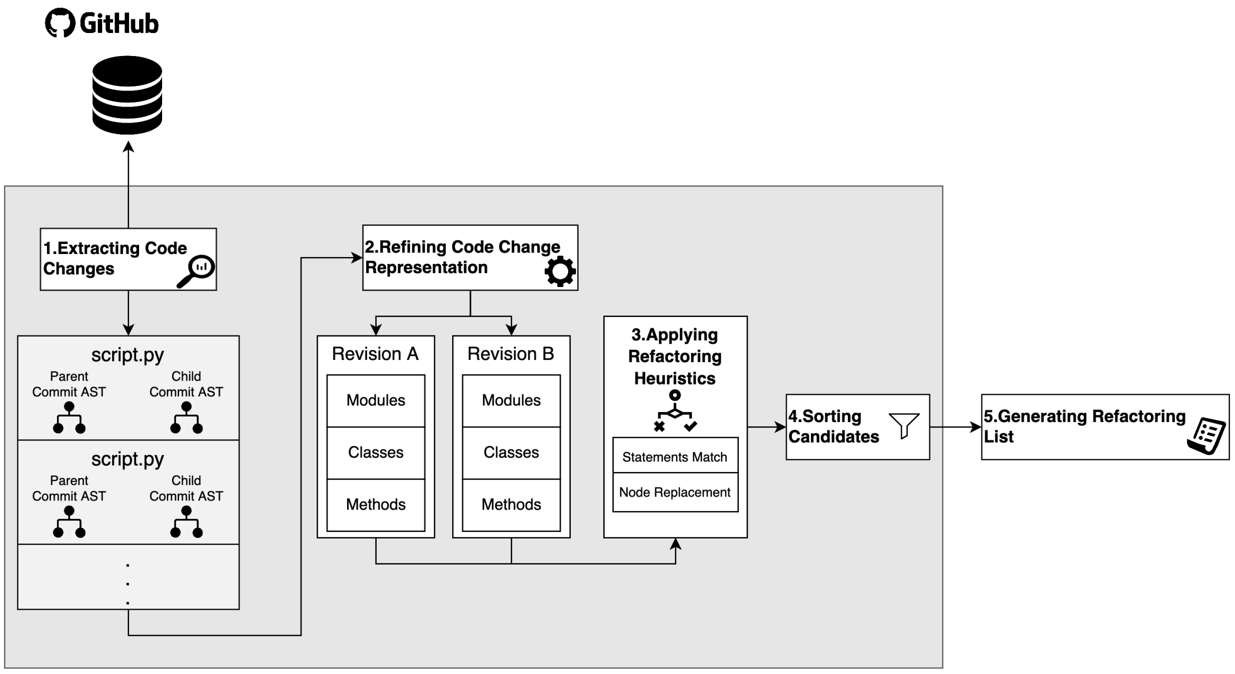


FIGURE 3.1: The Architecture of PYREF

TABLE 3.1: Examples of supported refactoring types

Refactoring Type	**Before Refactoring**	**After Refactoring**

Rename Method	`def methodA(self):`
	`    dummy()`
		`def methodB(self):`
		`    dummy()`

Change Parameter	`def methodA(self):`
	`    print("Hello␣World!")`
		`def methodB(self, _text):`
		`    print(_text)`

Change Return Type	`def methodA(self) -> lst[int]:`
	`    dummy()`
	`    return [1,0]`
		`def methodA(self) -> lst[bool`
		`    ↪ ]:`
		`    dummy()`
		`    return [True,False]`

Extract Method	`def methodA(self):`
	`    print("Hello␣World")`
	`    return [1,0]`
		`def methodA(self):`
		`    self.extracted_method()`
		`    return [1,0]`
		`def extracted_method(self):`
		`    print("Hello␣World")`

Inline Method	`def methodA(self):`
	`    self.to_inline_method()`
	`    return [1,0]`
	`def to_inline_method(self):`
	`    print("Hello␣World")`
		`def methodA(self):`
		`    print("Hello␣World")`
		`    return [1,0]`

Pull Up Method	`class classAChild(classA):`
	`    def methodC(self):`
	`        print("Hello␣World")`
	`    def methodB(self):`
	`        obj = a + c`
		`class classA:`
		`    def methodA(self):`
		`        dummy()`
		`    def methodB(self):`
		`        obj = a + c`

Push Down Method	`class classA:`
	`    def methodA(self):`
	`        dummy()`
	`    def methodB(self):`
	`        obj = a + c`
		`class classAChild(classA):`
		`    def methodC(self):`
		`        print("Hello␣World")`
		`    def methodB(self):`
		`        obj = a + c`

Rename Class	`class exampleClass:`
	`    def methodA(self):`
	`        dummy()`
		`class RenamedExampleClass:`
		`    def methodA(self):`
		`        dummy()`

Move Class	`# file util.py`
	`class exampleClass:`
	`    def methodA(self):`
	`        dummy()`
		`# file tool.py`
		`class exampleClass:`
		`    def methodA(self):`
		`        dummy()`

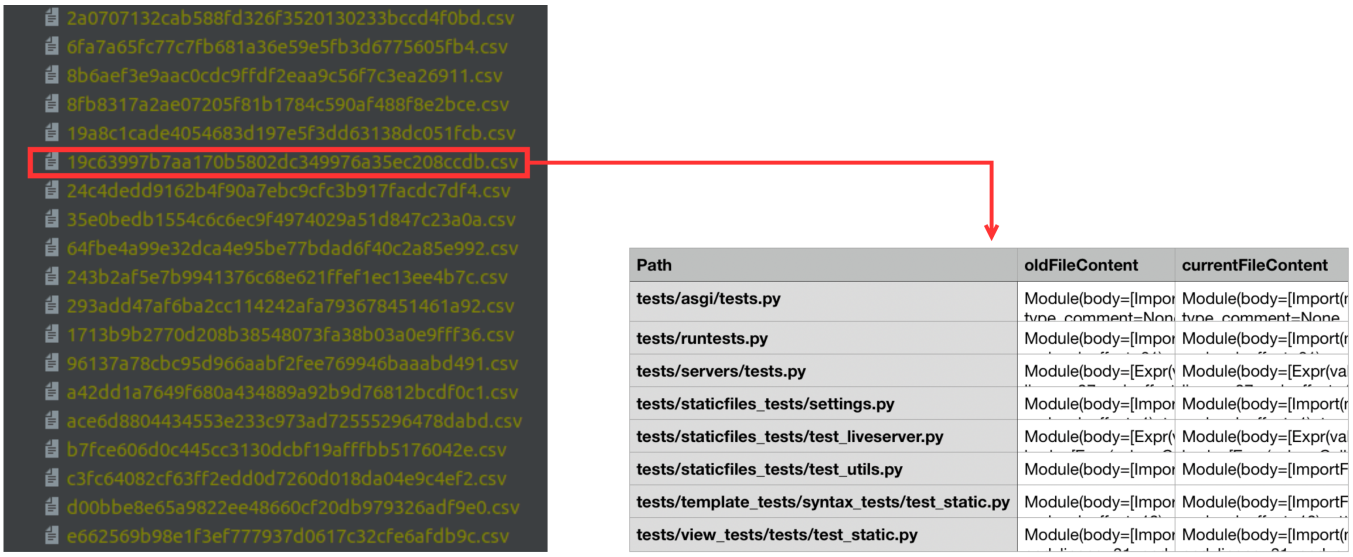
Extract Variable	`obj = obj1.func1().func2().`
	`    ↪ func3()`
		`ext_obj = obj1.func1()`
		`obj = ext_obj.func2().func3()`

While PYREF largely replicates the core refactoring detection algorithms of REFACTORINGMINER, some changes were applied due to the language differences between Java and Python. For example, one of the most prominent differences is that Python is dynamically typed, while Java is a statically typed language. This fact makes it impossible to directly adopting refactoring detection heuristics used by REFACTORINGMINER for PYREF. In addition to the adaptation caused by language differences, we have also slightly adjusted the code processing mechanism to enhance the performance and allow a more abstract process of information extraction, while REFACTORINGMINER extracts and stores a large amount of in-depth information regarding the code elements in the ASTs. Most of these changes and adjustments take place at stage 2 (*Refining Code Change Representation*) and the stage 3 (*Applying Refactoring Heuristics*).

3.2.1 Extracting Code Changes

In the *Extracting Code Changes* stage, PYREF locates and extracts the modified Python files among all of the commits in the history of a repository. More specifically, PYREF first uses GitPython library [2] to extract all the commit pairs $< commit_p, commit_c >$, where $commit_p$ and $commit_c$ represent the parent commit and the child commit, respectively. After that, like what REFACTORINGMINER does, PYREF only inspects the modified files between the $commit_p$ and $commit_c$. The main idea of singling out the modified files is to minimize the processing time and improve the performance, since refactorings only occur in a specific scope of the source code without affecting the whole repository. Including the source code of the whole project will only result in processing irrelevant files, which slows down refactoring detection, especially for large projects. As PYREF is only interested in the pairs of commits that include at least one modified Python file, all other commits are excluded from the proceeding stages. After locating the modified Python files in the commits, PYREF read and convert the source code of each modified file into a Python AST object using the build-in standard Python library *ast.py* [4]. Each modified file is associated with one pair of ASTs $<ast_p, ast_c>$, where ast_p and ast_c represent the AST objects of the Python file before and after the modification (*i.e.,* in $commit_p$ and $commit_c$), respectively.

The AST instance of a modified file contains the abstract grammar of the source code and is represented with a dictionary, a naive data structure in Python. In the AST, each item defines a code element and its attributes. The abstract grammar of a generic python code, as defined by ast.py, is organized with three hierarchical levels, namely *mod*, *stmt*, and *expr*. The *mod* level defines the module that the python code is hosted in; the *stmt* level contains all of the defined statements (*e.g.,* Assign Statement, Function Definition, etc.); and the *expr* level contains the expressions within the statements (*e.g.,* Invocation, Variable, Operations, etc.).



```
2a0707132cab588fd326f3520130233bccd4f0bd.csv
6fa7a65fc77c7fb681a36e59e5fb3d6775605fb4.csv
8b6aef3e9aac0cdc9ffdf2eaa9c56f7c3ea26911.csv
8fb8317a2ae07205f81b1784c590af488f8e2bce.csv
19a8c1cade4054683d197e5f3dd63138dc051fcb.csv
19c63997b7aa170b5802dc349976a35ec208ccdb.csv
24c4dedd9162b4f90a7ebc9cfc3b917facdc7df4.csv
35e0bedb1554c6c6ec9f4974029a51d847c23a0a.csv
64fbe4a99e32dca4e95be77bdad6f40c2a85e992.csv
243b2af5e7b9941376c68e621ffef1ec13ee4b7c.csv
293add47af6ba2cc114242afa793678451461a92.csv
1713b9b2770d208b38548073fa38b03a0e9fff36.csv
96137a78cbc95d966aabf2fee769946baaabd491.csv
a42dd1a7649f680a434889a92b9d76812bcdf0c1.csv
ace6d8804434553e233c973ad7255296478dabd.csv
b7fce606d0c445cc3130dcbf19afffbb5176042e.csv
c3fc64082cf63ff2edd0d7260d018da04e9c4ef2.csv
d00bbe8e65a9822ee48660cf20db979326adf9e0.csv
e662569b98e1f3ef777937d0617c32cfe6afdb9c.csv
```

Path	oldFileContent	currentFileContent
tests/asgi/tests.py	Module(body=[Impor type_comment=Non	Module(body=[Import(r type_comment=None
tests/runtests.py	Module(body=[Impor	Module(body=[Import(r
tests/servers/tests.py	Module(body=[Expr(	Module(body=[Expr(va
tests/staticfiles_tests/settings.py	Module(body=[Impor	Module(body=[Import(r
tests/staticfiles_tests/test_liveserver.py	Module(body=[Expr(	Module(body=[Expr(va
tests/staticfiles_tests/test_utils.py	Module(body=[Impor	Module(body=[ImportF
tests/template_tests/syntax_tests/test_static.py	Module(body=[Impor	Module(body=[ImportF
tests/view_tests/tests/test_static.py	Module(body=[Impor	Module(body=[Import(r

FIGURE 3.2: AST Pairs

TABLE 3.2: Examples of AST serialization

Statement	**AST Text Format**

`sum = total + 1`	`"Module(body=[Assign(targets=[Name(id='sum',␣ctx=Store())],`
	`value=BinOp(left=Name(id='total',␣ctx=Load()),␣op=Add(),␣right=`
	`↪ Constant(value=1,␣kind=None)),␣type_comment=None)],␣`
	`↪ type_ignores=[])"`

`if x > 3:`
` foo()`
	`"Module(body=[If(test=Compare(left=Name(id='x',␣ctx=Load()),␣ops=[`
	`↪ Gt()],␣comparators=[Constant(value=3,␣kind=None)]),␣body=[`
	`↪ Expr(value=Call(func=Name(id='foo',␣ctx=Load()),␣args=[],␣`
	`↪ keywords=[]))],␣orelse=[])],␣type_ignores=[])"`

`class exampleClass:`
`    def methodA(self):`
`        print("Hello")`
	`"Module(body=[ClassDef(name='exampleClass',␣bases=[],␣keywords`
	`↪ =[],␣body=[FunctionDef(name='methodA',␣args=arguments(`
	`↪ posonlyargs=[],␣args=[arg(arg='self',␣annotation=None,␣`
	`↪ type_comment=None)],␣vararg=None,␣kwonlyargs=[],␣`
	`↪ kw_defaults=[],␣kwarg=None,␣defaults=[]),␣body=[Expr(value=`
	`↪ Call(func=Name(id='print',␣ctx=Load()),␣args=[Constant(`
	`↪ value='Hello',␣kind=None)],␣keywords=[]))],␣decorator_list`
	`↪ =[],␣returns=None,␣type_comment=None)],␣decorator_list=[])`
	`↪ ],␣type_ignores=[])"`

To pass the AST pairs into the next stages, PYREF serializes the AST pairs of each modified file into a text representation and stores them into separated CSV files registered by the $commit_c$'s hash id. In Fig. 3.2, we can see the csv files on the left, where each csv file contains the modified python files along their ASTs pairs. Each file is represented by its path, and both of its ASTs in text representation (*i.e.,* oldFileContent, currentFileContent). The process of serialization is done by using the ast.dump() function provided by Python ast library. In Table 3.2, there are three given examples on how Python ASTs are serialized into a text representation.

The first row of the Table 3.2 consists of an AST node of type Assign (*i.e.,* stmt). As explained before, the *stmt* node consists of multiple *expr* nodes (*i.e.,* expressions). In our case, the assign statement in the first row consists of four *expr* nodes:

1. The first *expr* is a node representing the variable *sum*. This node has a type of *Name* with an *id* attribute indicating its given variable name (*i.e.,* identifier) and an *ctx* attribute indicating whether this variable is used to store or load value, in this case it is a store variable.

2. The second *expr* is a node representing the binary operation *total + 1*. This node has a type of *BinOp* with an *op* attribute indicating it is an addition. Besides, it also contains two *expr* nodes inside for the left and right operands of the binary operation.

3. The third *expr* is a node representing the variable *total*, which is the left operand of the addition. Like the variable *sum*, this node also has a type of *Name*, however, this variable is used to load value.

4. The fourth *expr* is a node representing the number *1*, which is the right operand of the addition. This node also has a type of *Constant* with a value of “1”.

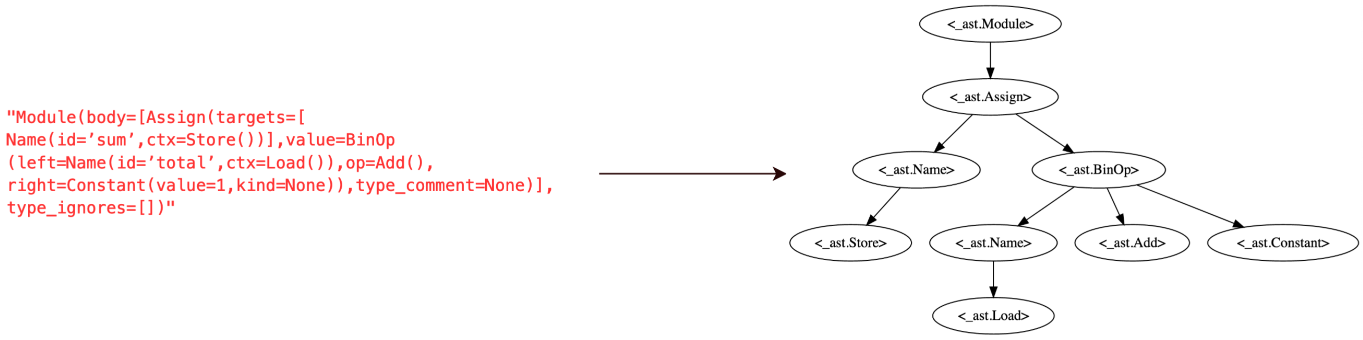
Similarly, the second row and the third row demonstrate how an if statement and a class are serialized, respectively. Each node in the AST has its own attributes, and some of these attributes can contain other nodes in a nested way (e.g., *body* attribute in the last two rows). In this format PYREF stores and loads the ASTs extracted from the modified python files.

3.2.2 Refining Code Change Representation

The stage of refining code change representation is responsible for loading the serialized AST, filtering out the unwanted elements (*e.g.,* docstrings), and finally organizing the code elements with a predefined model.

When PYREF loads the serialized text representation, it reconverts it to an AST object using the built-in method *eval*() [5], which takes a string representation of an AST and return an AST object. In this way, we get AST objects for both ast_p and ast_c, which contain all the code elements (*e.g.,* modules, classes, methods) in their associated Python file.

As the built-in AST library of Python is not very handy when it comes to navigation between AST nodes, we transferred each of the ASTs into a more customizable tree by using the library *anytree* [1] . For each node in the AST, we store it as a separate node and initiate edges between the node and its parent/child elements. With the help of *anytree*, we are able to easily navigate through nodes and edges. For example, to retrieve the parent of a certain node, we can simply use the *parent* attribute of the node (*i.e.,* `node.parent`). The ability to navigate with ease does not exist in the official AST library. Moreover, with *anytree* we can add or remove attributes of our choice for each node, and it is also possible to retrieve additional information alongside the AST object, such as index or depth of the node. One more important feature of the *anytree* library is the node searching ability by provoking the function *findall*(). With this function, we are able to search for any node in the tree with certain criteria. For instance, to find all variables in the tree with the an identifier "sum", we use the function *findall*() by passing it the type of node, and what value should its *id* contains. Fig. 3.3 illustrates how the textual representation of the AST object of the statement "`sum = total + 1`" can be converted into an *anytree* representation.



```
"Module(body=[Assign(targets=[
Name(id='sum',ctx=Store())],value=BinOp
(left=Name(id='total',ctx=Load()),op=Add(),
right=Constant(value=1,kind=None)),type_comment=None)],
type_ignores=[])"
```

FIGURE 3.3: Anytree Representation

After constructing trees for each AST object, we now have a pair of trees for each file $< tree_p, tree_c >$, representing the code before and after modification. PYREF extracts the needed code elements from each tree and represents them with a predefined model. We followed a similar information modeling approach as the one used by REFACTORINGMINER. The model includes the following elements:

- Module: A Python module is a file that contains definitions and statements. The module name is the same as the file name with the filename extension of ".py". For each tree representing the modified file, we extract its contained modules. The module is defined by its name, as well as contained classes and methods. The methods that are extracted here are those in the module but not affiliated to any class. That is, we do not include the methods defined within the classes.

- Class: Like modules, classes may hold definitions and statements as well. For each class in a module, we assign it with its name, a list of inherited classes (if any), the defined fields, and methods. The fields can be either instance variables or class variables, and they are extracted from the constructor of the class (*i.e.,* the "`__init__`" method) or any variable defined within the scope of the class only.

- Method: For each method, we associate it with the module and class that it belongs to, the name of the method, a list of parameters (if any), and the statements it carries. A method can have a null class, in case it was declared in a module directly.

- Leaf Statement: A leaf statement is a statement that does not contain nested statements inside. To detect if a statement is a leaf or not, we simply check if it has the attribute "body". A leaf statement contain all the elements it contains, such as variable, invocation, operators, constants, etc. In addition, leaf statement is associated with the parent method, its original AST in text representation, its depth, and the index in the tree. For example, we can consider the assign node in Fig. 3.3 as a leaf statement as it does not include the "body" attribute.

- Composite Statement: A composite statement contains the same attributes of a leaf statement. However, a composite statement is considered a statement with nested statements (*e.g.,* If statement in the second row of Table 3.2). The body of a composite statement can contain any type of statements. Therefore, composite statements, in addition to the attributes they have in common with leaf statements, may hold other leaf statements or composite statements.

While extracting the needed code elements, we filter out some elements such as ellipsis and docstring, since they have no significant impact on the detection of any type of refactoring. To facilitate the refactoring detection, PyRef also stores the information of the common, added, and removed code elements between the two commits. By categorizing the code elements into three groups (common, added, removed), we can have easier access to the changes that occurred. This information is vital for identifying some refactoring types with the help of defined heuristics that will be discussed in Section 3.2.3.

To detect whether an element in $commit_p$ has a match in $commit_c$, or an element is added or removed, we followed the same approach RefactoringMiner uses. The approach checks code elements of the same type from top to bottom level. In other words, we start by checking the modules, then moving to classes, and finally methods.

To identify the change status of modules (*i.e.,* unchanged, added, removed), we compare the names of the modules from both $commit_p$ and $commit_c$. If a module from $commit_p$ happens to have an equivalent name of another module in $commit_c$, we consider these two modules as matched, and add them to the common set. However, if a module in $commit_p$ has no module with an identical name in $commit_c$, we consider this module as a removed module, on the other hand if a module in $commit_c$ has no identical module in $commit_p$, we consider this module as an added module. We follow the same steps for classes. However, on the top of checking whether the names are similar, we also check if the modules that contains the class are the same. Finally to examine change status of methods, we start by checking if both candidate methods have the same module and class, list of parameters, method name, and return type (in case it exists). In Python, unlike Java, developers do not need to explicitly designate the return type of a method, since the return type will be decided during the execution. With that being said, we can not always determine whether the return types of two methods are the same. Therefore, if a method has no explicitly defined return type, we consider the return type as null. Moreover, since PyRef only performs static analysis of the code, it is not possible to determine the return type by observing the returned values. One way to tackle this problem is to consider the type of the returned nodes when there is no explicit return type given for the method, but this is inefficient and sometimes hard to capture in many cases, such as when the method has several returned nodes with different types. Below we summarize the conditions we use to determine the change type of code elements:

$$Module : module1.name = module2.name$$

$$Class : class1.name = class2.name \land class1.module = class2.module$$

$$\begin{aligned} Method : method1.name = method2.name \land method1.module = method2.module \\ \land method1.class = method2.class \land method1.params = method2.params \end{aligned}$$

With the three sets in hand (added, removed, and common), PyRef is ready to proceed to the next stage where it looks for detailed relationships among the elements of these sets. It is important to note that the obtained sets are not fixed and are subject to change depending on the outcome of the proceeding heuristics. For instance, if we are able to demonstrate through some heuristics that one class from $commit_p$ was renamed in $commit_c$, we can safely remove them from the added and removed sets and add them to the common set. It is very crucial to update the sets throughout the process of refactoring detection, because the refactoring heuristics heavily depend on these sets, and any update to them will impact the following refactoring detection. As in the previous example, if two classes are added to the common set, PyRef will be able infer more about possible refactoring inside the class. However, if the two classes are not added to the common set, PyRef will consider them as different classes and will not perform an in-depth search for method and variable level refactorings that might be contained in those classes.

3.2.3 Applying Refactoring Heuristics

The main goal of this stage is to apply a set of predefined heuristics that could detect the underlying refactorings in the source code. To apply these heuristics, PyRef needs to perform a more detailed examination regarding the differences among the three sets we obtained before. Unlike the stage of refining code change representation, in this stage we do not only compare the elements based on their signature, but also operate on the statement level to deduce any differences. More specifically, we rely on the statement matching algorithm proposed in RefactoringMiner [30]. The statement matching algorithm is one of the core components for refactoring detection and is responsible for matching statements within method bodies. The main advantage of this algorithm is that it does not only matches statements based on their textual similarity, but also is syntax aware. With the help of this approach, we are able to know how similar two methods are. For instance, if we applied this algorithm on a removed method and an added method, and get a larger number of matched statements than the unmatched statements, then we can indicate that these methods can be matched. With this piece of information and other heuristics, we can learn if they had undergone any refactoring process.

As can be seen in Algorithm 1, statement matching takes as input two different methods, and it returns as an output a sorted list of matched statements between the given methods. The first step of the algorithm is to extract leaf and composite statements from both methods and place them in two different lists. In the next step, the statements are preprocessed before being fed into the matching phase. The preprocessing phase consists of two steps: abstraction and argumentization.

Table 3.3: Example of abstraction and argumentization.

Refactoring Type	Before Refactoring	After Refactoring

Inline Method (abstraction)	`def methodA(self):`
	`    x = self.methodB()`
	`    print(x)`
	`def methodB(self):`
	`    return "Test"`
		`def methodA(self):`
		`  x = "Test"`
		`  print(x)`

Extract Method (argumentization)	`def methodA(self):`
	`    x = 1 + 1`
	`    print(x)`
		`def methodA(self):`
		`    self.methodB(1)`
		`    print(x)`
		`def methodB(self, p):`
		`    x = p + 1`

In the abstraction approach, we try to unify the statements of different AST node types as much as possible. The reason behind the abstraction is that some refactorings might apply fundamental changes to the statements. For example, refactorings such as Inline method often omit return statements and replace them with the expressions they held originally. As can be seen in the first row of Table 3.3, the inline refactoring has removed the return statement from methodB, and assigned the returned value "Test" directly to the variable x. In this case, we know that from the developer's point of view the statements `return "Test"` and `x = "Test"` are equivalent, but for the matching algorithm they look different. To deal with such cases, we omit the left side of the statement, and only consider the

Algorithm 1 Statement Matching

```
1: **Input** two adjacent methods m1 and m2
2: **Output** M = $\{(s_i, s_j)_1, ..., (s_i, s_j)_n\}$, sorted list of matched statements
3: M $\Leftarrow$ {}
4: $L_1 \Leftarrow m1.leafStatements$
5: $L_2 \Leftarrow m2.leafStatements$
6: StatementMatching($L_1, L_2$)
7: $C_1 \Leftarrow m1.compositeStatements$
8: $C_2 \Leftarrow m2.compositeStatements$
9: StatementMatching($C_1, C_2$)
10: procedure STATEMENTMATCHING($S_1$, $S_2$, M)
11:     for each $s_1 \in \mathcal{S}_1$ do
12:         for each $s_2 \in \mathcal{S}_2$ do
13:             preprocess$(s_1, s_2)$
14:             if $String(s_1) = String(s_2) \land Depth(s_1) = Depth(s_2)$ then
15:                 insert ($s_1$,$s_2$) into M
16:                 Break
17:             else if $String(s_1) = String(s_2)$ then
18:                 insert ($s_1$,$s_2$) into M
19:                 Break
20:             else if $NodeReplacer(s_1, s_2)$ then
21:                 insert ($s_1$,$s_2$) into M
22:                 Break
23:             end if
24:         end for
25:     end for
26:     M $\Leftarrow$ Sort(M)
27: end procedure
```

expression or the right side of the statement. In this specific example, the abstraction phase will have the following effect on the statements:

- return "Test" $\Rightarrow$ "Test"
- x = "Test" $\Rightarrow$ "Test"

The abstraction approach is always provoked in case of the presence of any of the following statement types: *"Return", "Raise", "Assert", "Import", "Delete", "ImportFrom", "Global", "Nonlocal", "Break", "Continue",* and *"Pass"*.

During the argumentization approach, we replace parameters with the initial values that was passed through the invocation arguments. For example, in the second row of Table 3.3, we can see that statement “x = 1 + 1” in the original “methodA” before the refactoring was parameterized to “x = p + 1” after applying the extract method refactoring. In this case, we substitute the parameter “p” in the statement “x = p + 1” of “methodB” with “1” (*i.e.,* “x = p + 1” in “methodB” is converted to “x = 1 + 1”) as “1” is passed to “methodB” during the invocation in the updated “methodA”. By applying this technique, we can increase the accuracy of the statement matching algorithm, and reduce its complexity.

After preprocessing each of the statements, we try to match leaf statements from the two lists L_1 and L_2 by checking if they satisfy one of the three conditions. In the first condition (line 13 in Algorithm 1), the statements have to be identical in their textual representation and have the same depth in their hosting tree. To meet the second condition (line 16 in Algorithm 1), it is enough for the statements to have an identical text representation without considering their depth in the tree. Finally in the last condition (line 19 in Algorithm 1), we make use of the algorithm NODEREPLACER, which tries to match statements by replacing their inner elements. If two statements satisfy one of these conditions they are removed from the original lists (L_1 and L_2) and added into the set of matched statements M.

In case of composite statements, passing one of the three conditions is not enough. They should also have at least one matched pair of statements within their bodies. After matching all statements, there might be some overlapping pairs in the output. In other words, one statement might get matched with two different statements. To address this issue, we group the statements, and sort them in ascending order based on three different attributes: their index in the body of their method, depth in the tree, and textual similarity. Then we only consider the first matched pair and discard the others.

In NODEREPLACER (Algorithm 2), we try to match the statements, which do not satisfy the first two conditions in Algorithm 1, and that is done in a less strict way by replacing their elements until they match. First, we extract the elements of each statement. The extracted elements are: variables, invocations, attributes, operators, and constants. For instance, the extracted elements of the following statements are:

- *"sum = total + 1"* ⇒ variables{*sum, total*}, operators {+}, and constants{*1*}.
- " sum = obj.att.func(2, p) " ⇒ variables{*sum, p,obj*}, constants{2}, attributes {*att*}, and invocations{*obj.att.func()*}.

Algorithm 2 NodeReplacer

```
1: Input two statements s1 and s2
2: Output M = $\{r_1, ..., r_n\}$, possible replacements
3: R ⇐ {}
4: $elements_1 \Leftarrow Sort(s1.getElements())$
5: $elements_2 \Leftarrow Sort(s2.getElements())$
6: NodeReplacer($s_1, s_2, elements_1, elements_2$)
7: procedure NODEREPLACER($s_1, s_2, elements_1, elements_2$)
8:     originalDistance = distance($s_1, s_2$)
9:     for each $e_1 \in elements_1$ do
10:        for each $e_2 \in elements_2$ do
11:            if $compatible(e_1, e_2)$ then
12:                alteredDistance = replace($e_1, e_2, s_1, s_2$)
13:                if $alteredDistance < originalDistance$ then
14:                    insert ($r_n$) into R
15:                    if $alteredDistance = 0$ then
16:                        Break
17:                    end if
18:                end if
19:            end if
20:        end for
21:    end for
22:    R ⇐ Sort(R)
23: end procedure
```

When extracting the elements, we sort them by their depth in descending order and index in ascending order. The reason of performing the sorting is to avoid the impact of outer elements on the inner elements. For example, when we are tying to match the two statements: *"x = a.b(p1,p2)"* and *"x = a.c(q1,p2)"*, if the elements were not sorted, it would have been enough to replace the invocation *"b(p1,p2)"* with *"c(q1,p2)"* to get identical statements. However, the replacement of the argument *"p1"* to *"q1"* would be ignored and thus not recorded. On the contrary, when the elements are sorted, we start from the bottom level of the elements then we move upward to reach the outer elements (*i.e.,* depth) and sustaining the direction from left to right (*i.e.,* index). In this way the replacement of the arguments “p1” with “q1” will not be missed, but will occur in the first place before replacing the entire invocation. In addition, no further inspections will need to be done for checking if any inner elements were skipped after the node replacement process. Sorting the inner elements is one of the adjustment we performed to the original algorithm of REFACTORINGMINER.

After extracting and sorting the elements of statements, we compute the original Levenshtein distance between the string representation of each statement. Before replacing elements with each others, we further check if those elements are compatible for replacement in term of syntax. For example, an operator can only be replaced with another operator, and variables that are meant to store values can only be replaced other variables storing values.

Another constraint to be considered when replacing elements is to only replace invocations that cover the entire statement with other invocations covering the entire statement. In order for an invocation to be considered covering the entire statement, the invocation should be governing the entire statement. For example, the statements `print("Hello World")` and `a = obj.call()` are considered invocations that cover the entire statements. If we allow these invocations to be replaced with all other types of elements, we might get an excessive number of matching statements. Therefore, invocations covering the entire statement are only allowed to be replaced by other invocations covering the entire statement. Moreover, we also apply additional constraints to avoid meaningless matching, namely the two invocations must 1) have an equivalent name; 2) have an identical list of parameters; or 3) satisfy both 1) and 2). When both the method names and the parameter lists are different, no replacement will be performed on the statements they belong to.

After checking if the replacement is what we needed, we proceed to perform the replacement on the code and compute the distance between the two statements before and after the element replacement. If the distance becomes smaller than the original computed distance after the replacement, we append the replaced elements in the replacement list R, and keep checking for other possible replacements.

In the last step, an overlapping issue might occur, as in the statement matching procedure one element can be matched with many other elements, and this is not valid in the refactoring. To fix this issue, we sort the redundant replacements based on the distance they can reduce, that is, we choose the one which can reduce the maximum distance. Finally, we aggregate all the possible replacements, and check if we can obtain two equivalent statements with these replacements.

In addition to the main goal of NODEREPLACER algorithm (*i.e.,* matching statements), we can also leverage the provided list of the possible replacements between statements to infer the type of refactoring. The information about how the elements within statements are replaced is important to detect low-level refactoring (*e.g.,* variable rename, inline variable, extract variable, etc.). The current version of PYREF supports the detection of extract variable refactoring.

In term of implementation, the NODEREPLACER algorithm has two variants. The first variant is based on AST, which performs the replacement of elements directly on the AST object, using the built-in class *ast.NodeTransformer*. However, this variant tends to be extremely slow with nested and complex statements and this can reflect very negatively on the overall performance of the tool since we overwhelmingly rely on the element replacement approach in the NodeReplacer algorithm. The unsatisfactory execution time is due to the mechanism of the class *ast.NodeTransformer*, which conducts a recursive search over all the elements in the statement when we try to replace or modify an element, thus the entire statement has to be traversed whenever we look for a single element. The second variant is string based, with this variant we perform the replacement on the string representation of the AST. In order to get the string representation of the AST, we use the built-in function ast.dump(), as seen in Table 3.1. With the string representation of the AST, we can modify or replace elements by performing some string manipulations. After that, we reconvert the string into its original AST format. The string based variant is much faster than the AST based one since no recursive search is needed as the replacements take place on the text instead of complex AST objects.

With the help of *Statement Matching* and *Node Replacer*, as well as other heuristics, we can infer some types of refactorings in the source code. For each refactoring type, we use a set of rules that simulate the possible taken steps by the developers when applying those refactorings. The rules are inspired by REFACTORINGMINER, and some of them were simplified. For each type of refactoring, we apply the following rules:

1. Change Method Signature: We first compare the methods from the removed and added sets obtained during the stage of refining code change representation and check if the methods from both sets share the same class and module. Then we examine whether the both methods have either all of their statements matched or a larger number of matched statements than the unmatched ones. Finally, with *compatibleSignatures* heuristic, we check whether the methods have a compatible signature by computing the intersections and differences between their lists of parameters. To know what has been exactly changed in the signature we inspect the differences in the elements of both method signatures (*i.e.,* name, parameters, return type).

$$\begin{aligned}(M, U_1, U_2) = StatementsMatching(method_1, method_2) | method_1 \in AddedSet \wedge method_2 \in RemovedSet \\ \wedge\ (method_1, method_2) \in class_i \wedge (method1, method2) \in module_i \\ \wedge\ ((method_1.unmatchedStatments = \varnothing \wedge method_2.unmatchedStatments = \varnothing) \\ \vee\ (M >= U_1 \wedge M >= U_2 \wedge compatibleSignatures(method_1, method_2)))\end{aligned}$$

$$\boxed{Rename\ Method : method_1.name \neq method_2.name}$$

$$\boxed{Change\ Method\ Return\ Type : method_1.returnType \neq method_2.returnType}$$

$$\boxed{Change\ Method\ Parameters : method_1.parameters \neq method_2.parameters}$$

$*\ M\ is\ the\ total\ number\ of\ matched\ statements$

$*\ U_1\ is\ the\ total\ number\ of\ unmatched\ statements\ in\ respect\ to\ method_1$

$*\ U_2\ is\ the\ total\ number\ of\ unmatched\ statements\ in\ respect\ to\ method_2$

$*\ compatibleSignatures \Rightarrow method_1.parameters \subseteq method_2.parameters \vee method_2.parameters \subseteq method_1.parameters \vee |method_1.parameters \vee |method_1.parameters \cap method_2.parameters| \geq |method_1.parameters \cup method_2.parameters| \backslash |method_1.parameters \cap method_2.parameters|$

2. Change Class Signature: We first compare the classes from the removed and added sets and check if there are any subsets between their fields and methods. If so, we perform two additional checks: 1) if the two classes have different names, then it is considered a rename class refactoring; 2) if the two classes have different modules, then it is considered a move class refactoring.

$$\begin{aligned}class_1 \in AddedSet \wedge class_2 \in RemovedSet \\ \wedge\ (class_1.fields \subset class_2.fields \vee class_2.fields \subset class_1.fields) \\ \wedge\ (class_1.methods \subset class_2.methods \vee class_2.methods \subset class_1.methods)\end{aligned}$$

$$\boxed{Rename\ Class : class_1.name \neq class_2.name}$$

$$\boxed{Move\ Class : class_1.module \neq class_2.module}$$

3. Extract Method: To identify extract method refactoring, we make use of methods from the common set and the added set. From the the common set we get a tuple of two common methods: $method_1$ from the $commit_p$ and $method'_1$ from $commit_c$ after the revision. From the added set, we get an added method $method_2$. Then we

check if the statements in $method_2$ exist in the body of $method_1$. Lastly, we check if $method'_1$ calls $method_2$, and if $method_1$ shares the same class with $method_2$.

$$\begin{aligned}(M, U_1, U_2) = StatementsMatching(method_1, method_2) | (method_1, method'_1) \in CommonSet \\ \wedge\, method_2 \in AddedSet \wedge method_1.class = method_2.class \wedge \\ \neg method_1.calls(method_2) \wedge method'_1.calls(method_2) \wedge M >= U_2\end{aligned}$$

$*$ *M is the total number of matched statements*

$*$ *U_1 is the total number of unmatched statements in respect to $method_1$*

$*$ *U_2 is the total number of unmatched statements in respect to $method_2$*

4. Inline Method: To identify inline method refactoring, similar to extract refactoring, we get a tuple of two common methods (*i.e., $method_1$* and $method'_1$) from the common set. However, instead of the added set we use the removed set, since the inlined methods get removed in the $commit_p$ to be integrated into another method from $commit_c$. We check if the statements in $method_2$ (*i.e.,* removed method) exist in the body of $method'_1$. Finally, we check if $method_1$ calls $method_2$, and $method'_1$ shares the same class with $method_2$.

$$\begin{aligned}(M, U_1, U_2) = StatementsMatching(method'_1, method_2) | (method_1, method'_1) \in CommonSet \\ \wedge\, method_2 \in RemovedSet \wedge method'_1.class = method_2.class \wedge \\ method_1.calls(method_2) \wedge \neg method'_1.calls(method_2) \wedge M >= U_2\end{aligned}$$

$*$ *M is the total number of matched statements*

$*$ *U_1 is the total number of unmatched statements in respect to $method'_1$*

$*$ *U_2 is the total number of unmatched statements in respect to $method_2$*

5. Move Method: To detect move method refactoring, as in the previous rules, we try to match methods from the removed and added sets. In addition to the previous conditions, we check whether the methods' containers (*i.e.,* module, class) are the same or not. In case both the methods reside inside in two different classes each, we can perform a further check on the status of the inheritance between their classes to determine if this is a pull up or push down refactoring.

$$\begin{aligned}(M, U_1, U_2) = StatementsMatching(method_1, method_2) | method_1 \in RemovedSet \\ \wedge\, method_2 \in AddedSet \wedge M > U_1 \wedge M > U_2 \wedge method_1.name = method_2.name \\ \wedge\, ((module_1, module'_1) \in CommonSet \vee (class_1, class'_1) \in CommonSet) \\ \wedge\, (method_1 \in class_1 \vee method_1 \in module_1)) \\ \wedge\, ((module_2, module'_2) \in CommonSet \vee (class_2, class'_2) \in CommonSet) \\ \wedge\, (method_2 \in class_2 \vee method_2 \in module_2))\end{aligned}$$

$$\boxed{Pull\ Up\ Method : isChild(method_1.class, method_2.class)}$$

$$\boxed{Pull\ Down\ Method : isChild(method_2.class, method_1.class)}$$

$*$ M *is the total number of matched statements*
$*$ U_1 *is the total number of unmatched statements in respect to* $method_1$
$*$ U_2 *is the total number of unmatched statements in respect to* $method_2$

6. Extract Variable: Unlike the rules for previous refactorings, in the case of extract variable, we operate on the statement level. With the help of the list of possible node replacements provided by the algorithm NODEREPLACER, we iterate over the pairs of replaceable nodes, and inspect if the second node of the tuple (*i.e.,* $node_2$) is a variable and contains the same value of the replaced node (*i.e.,* $node_1$). An example can be seen in the last row of Table 3.1, where the invocation obj1.func1() ($node_1$) is extracted into the new variable ext_obj ($node_2$).

$$node_1 \in statement_1 \wedge node_2 \in statement_2 \wedge replacement(node_1, node_2) \wedge isVariable(node_2) \wedge node_2.value = node_1.value$$

The heuristics of refactoring detection should be examined in the order listed above. While it might seem insignificant sometimes, the order of the heuristics might impact on accuracy of refactoring detection. Different types of refactorings are closely related to each others. In many cases, finding one refactoring might contribute to identify more refactorings that are based on it. As stated in REFACTORINGMINER paper [30], the chosen ordering is based on the distance a code element travels during the refactoring process. That is, we start by searching for the refactoring types that do not relocate the targeted code from its original place (*i.e.,* Rename Method, Rename Class), after that we search for the refactoring types that move code in a local scope (*i.e.,* Extract Method, Inline Method), and finally those types including moving code from its container such as a module or class to another container (*i.e.,* Move Method, Move class). To highlight the importance of this order, suppose that a rename method and an extract method refactoring are applied sequentially on the same method, if the order of heuristics is not obeyed (*i.e.,* detection of rename method first), PYREF will fail to detect the extract method refactoring. The tool will fail in detecting the extract method refactoring for the missing context of the rename method refactoring operation, where both methods (*i.e.,* $method_1$ and $method_2$) will not be matched and added to the common set after the detection of a rename method refactoring. With that being said, the heuristics behind detecting extract method refactoring operations depends on the common set which was not affected in the last round. Therefore, following the order of heuristics allows us to correctly infer more refactorings and reduce the false negative cases.

3.2.4 Sorting Candidates

In the stage of sorting candidates, we apply additional checks on the possible refactorings that we got from the previous stage. When applying the heuristics, each code element (*e.g.,* modules, classes, methods, etc.) might be associated with a number of refactoring instances. In practice, code elements can undergo only a certain amount of each refactoring type. For example, a method can be renamed only once in a commit, but it could be used to extract many methods. For this reason, in this stage for each code element we inspect their relationship with the number of associated refactorings, and we eliminate the invalid refactorings by prioritizing them based on similarity metrics. In this way, we can reduce false positives.

For refactoring types related to the change of method signatures (*i.e.,* Rename Method, Rename Parameters, Change Return Type), we only allow a one to one relationship (*i.e.,* a method signature can be changed once only). In case a method was assigned with multiple change signature refactorings, we take into consideration the similarity it has with its matched methods. The similarity is calculated based on two values: the sum of the string distance between their statements (*i.e.,* Levenshtein distance), and the number of statements that satisfied the conditions one and two in the statement matching algorithm 1. After computing these values, we can sort the refactorings accordingly in descending order and choose the best candidate and discard the others for not being valid.

In case of rename class and move class refactoring, we have a similar situation as the method signature refactoring, where one class can be renamed or moved only once in a span of two consecutive commits. As before we eliminate the invalid cases by relying on some similarities metrics, which are the number of fields and methods the matched classes share. For example, if we get as a result that there are two rename refactoring instances, in one instance classA was renamed to classB and in the other classA was renamed to classC. To decide which should be selected as the final output, we inspect the number of fields and methods classB and classC share with classA, and select the one with the higher similarity.

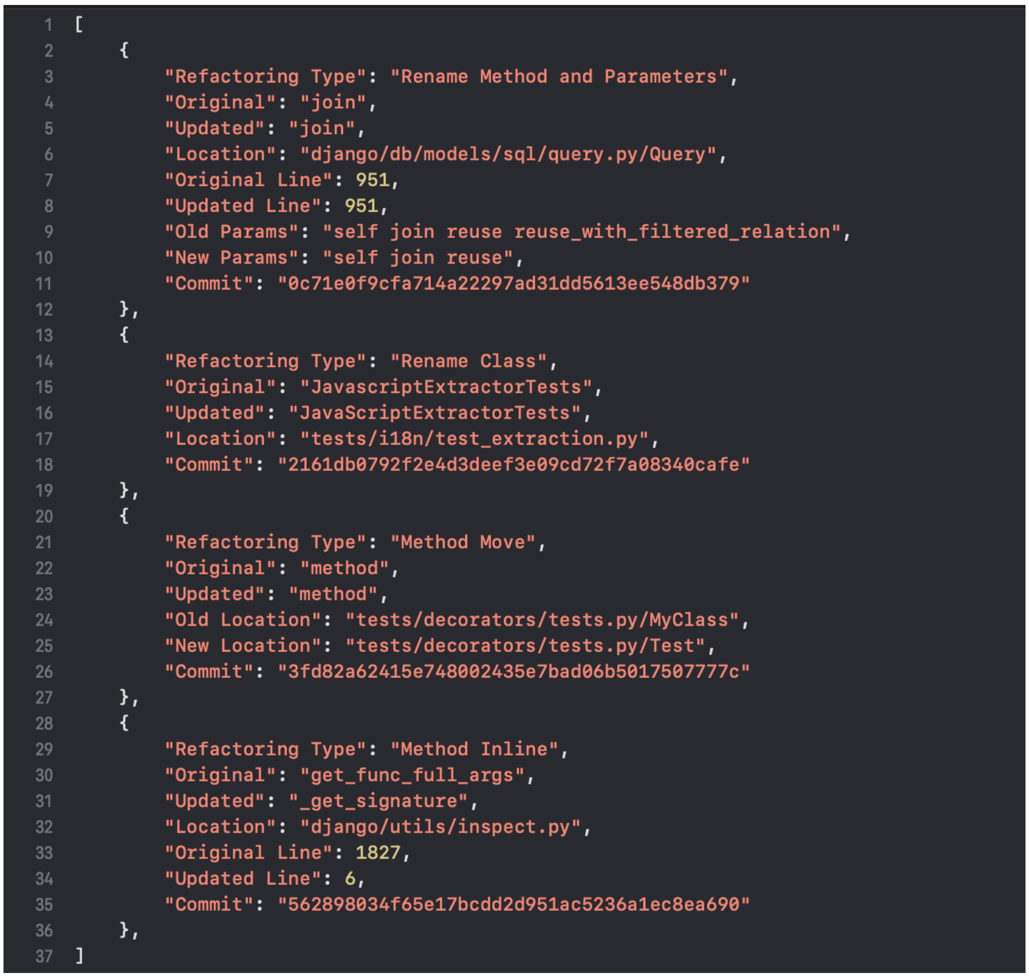
3.2.5 Generating Refactoring List

In the last stage, having all the possible refactorings in a repository or specific commits, we output the result in the JSON format. For each refactoring, we report the type, the associated commit, and the location where it occurs.

For example, Fig. 3.4 presents four possible refactorings detected in a repository. Each refactoring holds its own information. The fist refactoring is a Change Method Parameters operation, which was applied to a method named `join` contained in the class `Query` from the file “query.py”. In this refactoring, we can see the parameter $reuse\_with\_filtered\_relation$ was removed after the refactoring.

3.3 Summary

In this chapter, we thoroughly described the technical details behind PyRef. Given this tool, it is important to know whether it can detect refactoring operations accurately and efficiently. Therefore, in the next Chapter (Chapter 4), we will discuss how we conduct studies to verify the performance of PyRef.



```
[
    {
        "Refactoring Type": "Rename Method and Parameters",
        "Original": "join",
        "Updated": "join",
        "Location": "django/db/models/sql/query.py/Query",
        "Original Line": 951,
        "Updated Line": 951,
        "Old Params": "self join reuse reuse_with_filtered_relation",
        "New Params": "self join reuse",
        "Commit": "0c71e0f9cfa714a22297ad31dd5613ee548db379"
    },
    {
        "Refactoring Type": "Rename Class",
        "Original": "JavascriptExtractorTests",
        "Updated": "JavaScriptExtractorTests",
        "Location": "tests/i18n/test_extraction.py",
        "Commit": "2161db0792f2e4d3deef3e09cd72f7a08340cafe"
    },
    {
        "Refactoring Type": "Method Move",
        "Original": "method",
        "Updated": "method",
        "Old Location": "tests/decorators/tests.py/MyClass",
        "New Location": "tests/decorators/tests.py/Test",
        "Commit": "3fd82a62415e748002435e7bad06b5017507777c"
    },
    {
        "Refactoring Type": "Method Inline",
        "Original": "get_func_full_args",
        "Updated": "_get_signature",
        "Location": "django/utils/inspect.py",
        "Original Line": 1827,
        "Updated Line": 6,
        "Commit": "562898034f65e17bcdd2d951ac5236a1ec8ea690"
    },
]
```

FIGURE 3.4: Example of detected refactorings in the JSON format

Chapter 4

Evaluation

To *goal* of this study is to investigate (i) whether PYREF is able to detect refactoring operations accurately, and (ii) how efficient PYREF is when performing refactoring detection. The *context* consists of 27 refactoring examples collected from various online resources and 259 code changes detected as refactorings by PYREF. Besides, six projects are used to evaluate the execution time needed by PYREF.

4.1 Research Questions

Our study aims to answer the following research questions (RQs):

- RQ_1: *How reliable is* PYREF *in terms of detecting different types of refactoring operations?* One of the most important aspects to inspect when evaluating PYREF is whether the predicted refactorings are real ones. Ensuring reliable refactoring detection results can increase the confidence of users when using the tool, and it is essential for researchers to obtain convincing results when conducting relevant studies using PYREF. Therefore, this RQ aims to understand if PYREF can detect refactorings accurately.
- RQ_2: *How efficient is* PYREF *for detecting refactoring from Python Projects?*

  Another important aspect to consider during the evaluation is efficiency, for which the execution time often serves as a proxy. A relatively short execution time can help to a great extent to conduct large-scale studies on refactoring. Given the large number of repositories available online, fast refactoring detectors can lead to more generalizable and novel insights. Therefore, this RQ aims to understand how long much time PYREF needs to detect refactorings from Python repositories.

4.2 Context Selection & Data Collection

To answer RQ_1, we first built an oracle consisting of 27 refactoring operations extracted from three different online resources, which can be seen in Table 4.1.

TABLE 4.1: Online resources for documented refactoring instances

Source	**Total No. of Refactorings**
https://www.jetbrains.com/help/PYCHARM/refactoring-source-code.html	5
http://refactoring.encs.concordia.ca/oracle/statistics-pag	11
https://github.com/python-rope/rope	11
Total	27

To the best of our knowledge, so far there is no dataset available for evaluating the performance of refactoring detectors designed for Python. Given the fact that the refactoring process is rarely well documented, it is not trivial to collect refactoring instances from online resources. In this study, we mainly collect Python refactoring instances with the following three resources:

1. *Official website for* PYCHARM*[3].* Refactoring is one of the functionalities provided by PYCHARM IDE, which are documented with examples for some refactoring types. From the documentation we can retrieve five refactoring examples which can be used as oracles. These refactoring instances cover the following refactoring types: *Extract Refacoring, Move Method, Rename Method, and Pull Down Method.*

2. *Oracles used by* REFACTORINGMINER*[6].* The authors of REFACTORINGMINER provide a wide list of refactoring instances, however, as these refactorings were performed on Java, we had to manually rewrite the Java refactoring examples in Python. Due to the fact that translating the programming language manually can be very time consuming and sometimes even impossible, in the end we selected 11 cases from the oracle of REFACTORINGMINER and rewrote them in Python. The 11 cases covers the following refactoring types: *Inline Extract, Extract Refacoring, Move Method, Rename Method, Push Down Method, and Class Rename.*

3. Python refactoring library ROPE[7]. ROPE is a Python library to help developers perform refactoring operations on Python code. Their test cases illustrate many examples the expected code after applying certain refactorings. We extracted these examples to evaluate our tool, and the examples cover the following refactoring types: *Inline Extract, Extract Refacoring, Rename Method, Rename Parameters, and Move Method.*

With these obtained refactoring operations, we could measure how reliable PYREF is in a small scale. As this dataset contains only refactoring instances, there would be no false positive cases (*i.e.,* reporting non-refactoring operations as refactoring). Therefore, we report for how many cases PYREF can correctly detect the refactoring and for how many cases it fails to do so. We also report the value of recall.

Given the fact that we could only collect a few refactoring samples online, we also conducted case studies on 10 real Python projects and manually inspect whether they are real refactorings. The selected projects are listed in Table 4.2. Our tool reported 259 code changes as refactorings, which cover all the supported refactoring types except *Class Move* and *Extract Variable*. We manually inspected each refactoring operation to determine whether it is true or false positive. As no pre-labeled data available, in this case there would be no false negatives. Therefore, we report the numbers of true positive and false positive, as well as the precision. Given the relative large number of refactorings in this dataset, we can have a more intuitive idea to what extent we can trust the refactorings reported by PYREF. It is important to note that for some repositories we did not run the tool on the entire commits history but only on a chunk of them, as some repositories contain an immense number of commits which might take the tool a considerably long time to analyze.

TABLE 4.2: Selected repositories for manually inspecting the refactoring detection results

GitHub repository	**Total No. of Refactorings**
ethereum/py-evm	7
erpalma/throttled	2
getpelican/pelican-plugins	3
fossasia/susi_linux	1
faif/python-patterns	8
Yggdroot/LeaderF	30
not/kennethreitz_envoy	22
ekzhu/datasketch	68
django/django	75
damnever/pigar	43
Total	259

To answer RQ_2, we randomly selected six repositories with varying number of commits to measure the time execution of PYREF. The selected repositories can be seen in Table 4.3, with the number of commits and lines of code. After cloning these repositories, we ran our tool on them and recorded the execution time. The execution time was reported by the built-in Python library CPROFILE, which profiles Python programs and report its execution time with the unit of seconds.

TABLE 4.3: Selected repositories for execution time examination

Source (GitHub repository name)	**Total No. of Commits**	**Lines of Code**
xinntao/BasicSR	368	7890
cr0hn/dockerscan	59	1920
pypyjs/pypyjs	361	1284
swisskyrepo/SSRFmap	89	1322
miguelgrinberg/microblog	24	1264
mbi/django-simple-captcha	373	1547
Average	212.3	2537

4.3 Results

4.3.1 RQ_1: How reliable is PYREF in terms of detecting different types of refactoring operations?

Results for oracles obtained from online sources

Table 4.1 reports the numbers of true positives (*i.e.,* refactorings correctly detected by the tool) and false positives (*i.e.,* refactorings not detected by the tool) for each type of refactorings in the oracle collected from three online resources. In total, for the 27 refactorings, PYREF was able to detect 21 of them, leading to a recall of 77.8%. To have a better understanding on when and how our tool failed to detect refactorings, we illustrate some representative cases. These cases can help us to improve the heuristics used by the tool.

TABLE 4.4: Refactoring detection results for oracles collected from online resources.

Refactoring Type	**TP**	**FN**
Rename Method	2	0
Change Method Parameters	1	0
Move Method	1	2
Push Down Method	1	1
Extract Method	15	1
Inline Method	1	1
Rename Class	0	1
Total	21	6

One example of false negative can be seen in Table 4.5. This refactoring is a Move Method operation documented in the official PYCHARM website. The author of the refactoring moved the method `super_method` from the class `SuperClass` into two other classes (*i.e.,* `SubClassOne` and `SubClassTwo`). However, PYREF could only detect one of these two Move Method refactorings (*i.e.,* from `SuperClass` to `SubClassOne`), while missing the other one (*i.e.,* from `SuperClass` to `SubClassTwo`). The reason behind the failed detection is that we limited the number of refactoring instances for the Move Method refactoring, as discussed in Section 3.2.4. Currently, PYREF only allows a one to one matching for Move Method refactoring, namely a method can only be moved once in one commit. Therefore, PYCHARM is not able to report the method move from `SuperClass` to `SubClassTwo`.

Although this issue might seem trivial to fix, allowing a one to many relationship of Move Method refactoring might increase the chance of reporting false positive cases.

TABLE 4.5: First Example of false negative in the oracle collected from online resources.

Before refactoring:

```
class SuperClass:
   def super_method(self):
      pass
class SubClassOne(SuperClass):
   def my_method(self):
      pass
class SubClassTwo(SuperClass):
   def my_method(self):
      pass
```

After refactoring:

```
class SuperClass:
   pass
class SubClassOne(SuperClass):
   def my_method(self):
      pass
   def super_method(self):
      pass
class SubClassTwo(SuperClass):
   def my_method(self):
      pass
   def super_method(self):
      pass
```

Another interesting false negative case can be seen in Table 4.6. This case is a Rename Class refactoring originally from the oracle of REFACTORINGMINER and it was manually translated from Java to Python by us. In this refactoring, the author intended to rename the inner-class `TrailingHttpHeadersNameConverter` to `TrailingHttpHeadersNameValid`. PYREF was not able to detect such case due to the small number of shared elements between these two classes. As discussed in Section 3.2.3, to identify a code change as Rename Class refactoring, the matched classes must share a subset of methods and fields. In this example, the class, before renaming, has two methods and two fields, while the method after refactoring has removed one field and two fields. Indeed, refactoring operations are often performed together with other code changes. Sometimes, the unrelated changes might negatively impact the performance of refactoring detection tools if the original code was heavily modified.

Results for case studies on 10 projects

Table 4.7 reports the numbers of truth positives (*i.e.,* refactorings correctly detected by the tool) and false postives (*i.e.,* non-refactorings reported as refactorings by the tool) for each type of refactorings in the case study involving 10 real Python projects. In total, for the 257 reported refactorings by PYREF, 230 of them are true positive, leading to a precision of 89%. The high precision indicates that our tool actually can generate reliable results. However, we do notice that it performs relatively bad on some specific refactoring types including Move Method and Extract Method. Similar to what we have done before, we illustrate some representative cases to gain a better understanding on when our tool works well and when it fails to detect refactorings.

One example of false positive can be seen in Table 4.8. Our tool reported there was a Inline Method refactoring during the code change, while in reality it did not happen. In this case, we can find that before the code change, the method `share_post` invokes the method `article_summary` within its body. While after the code change, the method `share_post` does not call the method `article_summary` as the latter got removed from the source code entirely. This fact indicates a high chance of an Inline Method refactoring operation according to the heuristics.

TABLE 4.6: Second Example of false negative in the oracle collected from online resources.

Before refactoring:

```
class TrailingHttpHeaders:
   class TrailingHttpHeadersNameConverter:
      def TrailingHttpHeadersNameConverter(validate):
         super(validate)

      def convertName(name):
         name = super.convertName(name)
         if validate:
            if (HttpHeaderNames.CONTENT_LENGTH.equalsIgnoreCase(name)):
               raise IllegalArgumentException("prohibited␣trailing␣header:␣" + name)
         return name

      VALIDATE_NAME_CONVERTER = TrailingHttpHeadersNameConverter(true)
      NO_VALIDATE_NAME_CONVERTER = TrailingHttpHeadersNameConverter(false)

   def TrailingHttpHeaders(validate):
      super(validate, VALIDATE_NAME_CONVERTER)
```

After refactoring:

```
class TrailingHttpHeaders:
   class TrailingHttpHeadersNameValid:
      def TrailingHttpHeadersNameConverter(validate):
         INSTANCE = TrailingHttpHeadersNameValidator()

      def validate(name):
         HeaderNameValidator.INSTANCE.validate(name)
         if (HttpHeaderNames.CONTENT_LENGTH.equalsIgnoreCase(name)):
            raise IllegalArgumentException("prohibited␣trailing␣header:␣" + name)

   def TrailingHttpHeaders(validate):
      super(validate, VALIDATE_NAME_CONVERTER)
```

TABLE 4.7: Refactoring detection results for case studies on 10 projects.

Refactoring Type	**TP**	**FP**
Rename Method and Parameters	3	1
Rename Method	38	0
Rename Parameters	136	2
Move Method	15	9
Inline Method	4	0
Extract Method	21	11
Class Rename	13	4
Total	230	27

However, to determine whether it is indeed an Inline Method operation, there is another condition to meet. That is, the tool must find some matched statements between the method `article_summary` in the original code and the method `share_post` after the code change. Therefore, our tool must have mismatched some statements in these two methods. More detailed inspection is needed to understand what contributes to the mismatching

By looking back at Table 4.1 and Table 4.2, we can notice that Extract Method and some refactorings related to method signature changes related have the highest numbers of true positive cases. The reason is two-fold: First,

TABLE 4.8: Example of false positive in the case study involving 10 projects.

Before refactoring:

```
def article_summary(content):
   return quote(BeautifulSoup(content.summary, 'html.parser').get_text().
      ↪ strip().encode('utf-8'))
def share_post(content):
   if isinstance(content, contents.Static):
      return
   title = article_title(content)
   url = article_url(content)
   summary = article_summary(content)
   tweet = ('%s%s%s' % (title, quote('␣'), url)).encode('utf-8')
   diaspora_link = 'https://sharetodiaspora.github.io/?title=%s&url=%s' % (
      ↪ title, url)
   content.share_post = share_links
```

After refactoring:

```
def share_post(content):
   if isinstance(content, contents.Static):
      return
   title = article_title(content)
   url = article_url(content)
   tweet = ('%s%s%s' % (title, quote('␣'), url)).encode('utf-8')
   diaspora_link = 'https://sharetodiaspora.github.io/?title=%s&url=%s' % (
      ↪ title, url)
   content.share_post = share_links
```

these refactoring types are more frequently applied across different repositories; Second, with the high number of occurrences, we were able to study many cases and adjust the heuristics until we got a high accuracy in detecting these specific types of refactorings.

As our tool has successfully collected a relatively large number of real refactoring operations, here we illustrate a case in Table 4.9. In this example, the method `getwrapped` was extracted from the method `makebold` and `makeitalic`. This successfully detected case proved that several core algorithms behind PyRef functions correctly. As can be observed in the code before refactoring, both `makebold` and `makeitalic` contain an inner method `wrapped`, and these two inner methods share the same name but with different return values. In the code after refactoring, these inner methods were removed from `makebold` and `makeitalic`, and a new method `getwrapped` was added. While these specific changes indicate a high chance of an Extract Method operation, the tool still needs to check if the newly added method (*i.e.,* `getwrapped`) shares some statements with both `makebold` and `makeitalic` in the original code.

During the execution of the statement matching algorithm, two preprocessing approaches (*i.e.,* abstraction and argumentization) were applied. Starting with the argumentization approach, as mentioned before, some refactoring operations parameterize some variables in the process of refactoring. In this example, we can see that the characters "b" and "i" were replaced by the parameter *tag*. 0ur tool in this case can replace `tag` with `"b"` in the method `getwrapped` before matching this statement with statements in the method `makebold`. Similarly, it can also replace `tag` with `"i"` when trying to find a match in the method `makeitalic`. Next PyRef applied abstraction. Since most of the statements in this example are return statements, after abstraction, the node of return is discarded and only the returned values were kept. For example, the statement `return "<b>" + fn() + "</b>"` was

TABLE 4.9: Example of true positive in the case study involving 10 projects.

Before refactoring:

```
def makebold(fn):
    @wraps(fn)
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped
def makeitalic(fn):
    @wraps(fn)
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped
```

After refactoring:

```
def makebold(fn):
    return getwrapped(fn, "b")
def makeitalic(fn):
    return getwrapped(fn, "i")
def getwrapped(fn, tag):
    @wraps(fn)
    def wrapped():
        return "<%s>%s</%s>" % (tag, fn(), tag)
    return wrapped
```

converted into `"<b>" + fn() + "</b>"` before statement matching. This reduces complexity and noises when applying the statement matching algorithm.

After the preprocessing phase is completed, the tool started to match the statements. In this example we got a satisfying number of matched statements, therefore our tool can infer that this is an Extract Method operation.

4.3.2 RQ_2: How efficient is PYREF for detecting refactoring from Python Projects?

Table 4.10 presents the recorded execution time for each of the selected repositories, along with the number of detected refactoring operations. The execution time needed by these repositories varies from 23 seconds to slightly more than nine minutes, with an average of less than three minutes.

TABLE 4.10: Execution time of PYREF on six projects.

Repository	Execution Time (Sec)	Found Refactorings
xinntao/BasicSR	551.0	138
cr0hn/dockerscan	23.78	14
pypyjs/pypyjs	122.29	10
swisskyrepo/SSRFmap	22.9	5
miguelgrinberg/microblog	9.6	1
mbi/django-simple-captcha	205.5	32
Average	155.845	33

By combining with Table 4.3, we can find that overall the execution time is positively correlated to the number of commits. However, while the project `xinntao/BasicSR` and `pypyjs/pypyjs` have a similar amount of commits (368 vs 361), the former takes more than four times of time compared to the latter. The reason could be that lines of code play a role here, as the former is much larger in terms of size (with an LOC of 7890 compared to an LOC of 1284 for the latter). As mentioned before, the core approach of PYREF operates on the statement level by attempting almost every possible combination of statement nodes until the statements match. Therefore, more statements involved will lead to significant increase of the execution time.

It is important to note that we used the string based variant of the statement matching algorithm to evaluate the execution time. The string based variant is faster than the AST based variant. To highlight the difference in respect of execution time between these two variants, we ran the the tool separately with the AST and string based implementation on the first repository in Table 4.10 (*i.e.,* xinntao/BasicSR). The results shows that the AST based variant takes 845.6 seconds, while the string based variant takes 551 seconds.

4.4 Discussion

Our evaluation indicates that PYREF is able to detect refactorings accurately and efficiently. Therefore, PYREF can help researchers get reliable results when conducting software engineering studies involving refactoring instances in Python projects. despite of the good performance, our evaluation indicates that PYREF does not work well in some cases, especially when the refactoring is mixed with other code changes. It would be interesting to look into which types of other changes have more impact on the reliability of PYREF. Studying the false positive and false negative cases will provide us with the necessary information to differentiate refactoring operations from other code changes and further improve our tool. However, we acknowledge that due to the complexity of code changes, heuristics rules cannot be exhaustive, and there might always be some specific refactoring operations that are not bound to the rules.

Another valuable information that can be derived from the results is that certain relationships exist between different refactoring operations. For example, developers might tend to apply refactoring operations in a specific order, and some refactoring operations are often applied together. If we can integrate this information into our heuristics, we might be able to improve the accuracy of refactoring detection. For example, often different rename refactorings (*i.e.,* Rename Class, Rename Method, Rename Variable) tend to take place simultaneously. We have observed in some studied cases that when developers applied a Rename Class refactoring operation, the inner methods of that class were also renamed. Knowing this fact, we can give more weights to the Rename Method refactoring at the stage of sorting candidates if its associated class is also renamed. This adjustment might help us to detect refactorings more accurately.

A large number of the refactoring operations reported by PYREF were not pure refactorings. In fact, they had other behavioral changes included. However, PYREF, thanks to the adopted approach, can highlight and set apart the changes from the refactoring operations. This is essential to study the impact of refactorings on non-functional aspects, such as performance and energy consumption. Without cleaning the data by removing non-refactoring changes performed together with refactorings in code changes, the validity of obtained results will be highly questionable.

4.5 Threats to validity

In this section, we discuss the threats to validity that exist in our performance evaluation.

Threats to construct validity concern the relation between the theory and the observation, and in this work they are mainly due to the measurements we performed. In one part of our evaluation, the performance is evaluated on the refactoring instances from online resources. These refactorings are often artificial and built for some specific purposes, such as testing and documentation. Therefore, they might not reflect how refactoring is done in practice.

Currently, PYREF is the only tool that can detect refactorings in python repositories, thus it is impossible to compare it with other approaches. Therefore, we can only perform the evaluation based on the new oracles we collected in this study.

Threats to external validity concern the generalization of our findings. PYREF currently supports the detection for 11 types of refactorings. Given the fact that there are much more refactoring types, our current evaluation cannot prove that heuristics-based approach also works well for other types of refactorings. However, as REFACTORINGMINER 2.0, which inspires PYREF, supports over 40 refactoring types, we believe that PYREF will still be able to achieve an acceptable evaluation for newly added refactoring types in the future.

Another threat is that the size of our oracle is not huge. We obtained the documented refactorings from three online resources, which are far from comprehensive. However, given the fact that refactorings are rarely documented by developers, building an oracle would require significant effort for manual inspection.

When we conducting case studies, we only randomly selected 10 projects, and we used 6 projects for evaluating the efficiency of the tool. The choice is based on the fact that evaluating our tool on more projects would require much more time and more human resources for manually examining the results. We would like to extend our evaluation in the future.

Threats to internal validity concern internal factors of our study that could hinder its validity. As can be seen from the results, the current evaluation does not include some of the supported refactoring types (*i.e.,* Class Move, Extracted Variable). That is because these refactoring types rarely occur in the tested repositories, which is a known issue for other refactoring detectors when evaluating the performance.

4.6 Summary

Overall, the conducted evaluation has proven that the tool is qualified with an acceptable degree of reliability and efficiency. During the evaluation, we have built our own oracle, which is the first of the kind for refactoring detection in Python projects. However, we acknowledge that the size of our oracle is not huge, expanding the oracle will result in a more convincing evaluation which in turns will enhance the performance of the tool.

Chapter 5

Conclusions and Future Work

5.1 Recap

To fill the gap of missing refactoring detection tool for the Python language, we developed PyRef, which highly relies on the algorithms behind RefactoringMiner with some necessary adjustments.

The tool was evaluated against a manually crafted oracle including 27 refactoring instances collected from three different online resources, and 259 manually inspected code changes detected as refactorings by PyRef. We also test the execution time of PyRef on six different Python projects. Our results indicate that our tool can detect refactorings from Python projects reliably and efficiently.

5.2 Reflections

During the development and evaluation of PyRef, we have established a deeper knowledge in dealing with refactoring detection approaches. For example, during source code information extraction (stage 2 in Fig. 3.1), refactoring detection often requires very detailed information about how code elements are changed. As mentioned in Section 3.2, our information modelling of the code elements is more abstract than that used by RefactoringMiner, which not only contains the information regarding the high level code elements (*e.g.,* Moduels, Classes, Statements), but also include detailed information of various low-level elements (*e.g.,* Invocation, Variables). If we could store the in-depth information also for low-level code elements, the refactoring detection could be more accurate and precise.

Another core concept that can be added to the gained knowledge is the effectiveness of the predefined rules approach (*i.e.,* heuristics) over other approaches such as similarity based approaches. Like RefactoringMiner, we adopt the heuristics based approach, which do not thresholds required by similarity based approaches during statement matching. This approach has been proved quite powerful, and we do not need to rely on empirical values to obtain the threshold, which might change for differet types of projects.

Lastly, during the development of PyRef, we had to be aware of the significant differences between the languages of Java and Python. Since the used approach was initially designed to work with Java code, we had to make some necessary adjustment, for example, when dealing with the fact of the absence of types in Python.

5.3 Future Work

The tool holds many potentials and provides the necessary infrastructure for the future work. Some of the possible future work ordered by priority are listed below:

- One of the most important future work that can be done is the enlargement of the oracle by extracting more real refactoring cases with more inspection done by refactoring researchers. With a robust oracle the tool can be re-calibrated accordingly to generate more accurate results.
- We would also like to improve the performance of the sorting candidates stage in the architecture. Currently most of the false positive cases are due to a faulty behavior during this stage, and adjusting how refactoring candidates get sorted will enhance the accuracy of the tool.

- Another significant future work is adding more supported refactoring types especially Python specific refactorings. For now PYREF only supports the generic refactoring types, but more Python related refactorings can be added such as the transformation of loops into list comprehensions.
- Our future work can also include a visualization of the output in the form of diagrams and abstract graphs. Visualizing the output can help in interpreting each case more easily. Currently users have to read the JSON file to interpret what is happening.

5.4 Final Words

We presented PYREF, a tool designed to mine refactoring cases in Python repositories. Our tool adopt the core algorithms and heuristics of REFACTORINGMINER with few modifications. Thanks to this approach, the tool is able to detect low-level and high-level refactorings in specific commits. Even though the evaluation is based on a relatively small number of cases, we could prove that this approach is applicable on Python projects and is able to extract valuable refactoring instances which can be used in future studies.

Bibliography

[1] anytree. https://anytree.readthedocs.io/en/latest/.

[2] Gitpython. https://github.com/gitpython-developers/GitPython.

[3] Pycharm. https://www.jetbrains.com/pycharm/.

[4] Python ast. https://docs.python.org/3/library/ast.html.

[5] Python builtin eval. https://docs.python.org/3/library/functions.html#eval.

[6] Refminer oracle. http://refactoring.encs.concordia.ca/oracle/.

[7] Rope a python refactoring library. https://github.com/python-rope/rope.

[8] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[9] G. Antoniol, M. Di Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. pages 31 – 40, 10 2004.

[10] S. Davies, M. Roper, and M. Wood. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process*, 26(1):107–139, 2014.

[11] K. De Volder. *Type-oriented logic meta programming*. PhD thesis, Citeseer, 1998.

[12] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pages 404–428, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[13] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007.

[14] M. Fowler. *Refactoring: improving the design of existing code (2nd Edition)*. Addison-Wesley Professional, 2018.

[15] H. C. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. *IEEE software*, 26(1):26–33, 2009.

[16] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[17] A. Ketkar, N. Tsantalis, and D. Dig. Understanding type changes in java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 629–641, New York, NY, USA, 2020. Association for Computing Machinery.

[18] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 371–372, New York, NY, USA, 2010. Association for Computing Machinery.

[19] A. Loh and M. Kim. Lsdiff: A program differencing tool to identify systematic structural differences. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, page 263–266, New York, NY, USA, 2010. Association for Computing Machinery.

[20] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. pages 252–266, 01 2007.

[21] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[22] E. C. Neto, D. A. da Costa, and U. Kulesza. The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390, 2018.

[23] W. F. Opdyke. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proc. SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[24] S. Raschka. *Python machine learning*. Packt publishing ltd, 2015.

[25] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 858–870, New York, NY, USA, 2016. Association for Computing Machinery.

[26] D. Silva and M. T. Valente. Refdiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279, 2017.

[27] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, pages 10–10, 2007.

[28] L. Tan and C. Bockisch. A survey of refactoring detection tools. pages 1–1, 2019.

[29] N. Tsantalis, A. Ketkar, and D. Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020.

[30] N. Tsantalis, A. Ketkar, and D. Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020.

[31] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 126–130, 2003.

[32] Z. Xing and E. Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 54–65, New York, NY, USA, 2005. Association for Computing Machinery.

[33] Z. Xing and E. Stroulia. The jdevan tool suite in support of object-oriented evolutionary development. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, page 951–952, New York, NY, USA, 2008. Association for Computing Machinery.