



# Automated Clone Elimination in Python Tests

Sebastian Kingston<sup>1</sup>, Violet Ka I Pun<sup>2</sup>, and Volker Stolz<sup>1,2</sup>

<sup>1</sup> University of Oslo, Norway  
{smkingst,stolz}@ifi.uio.no

<sup>2</sup> Western Norway University of Applied Sciences, Norway  
{vpu,vsto}@hvl.no

**Abstract.** Code clones are a well-known software quality metric with existing tools for detection and (semi-) automated elimination for common programming languages. While they are usually eliminated by extracting duplicate code into shared methods/functions, we are here looking in particular at Python code for test cases, where clones stem from repeated test cases with primarily different arguments and expected results. In this scenario, the ideal solution is not introducing shared code, but rather using parametrized tests from the unit testing framework. We combine an existing clone detector (NiCad) with our own code transformation that eliminates code clones in Python test cases using the `pytest` framework. We show the usefulness of our approach by surveying open source Python projects that can benefit from our refactoring and evaluate the performance and correctness of our transformation by comparing unit-test results before and after.

## 1 Introduction

Testing plays an indispensable role in modern software development. A common way of writing a test is by copy-pasting another test, then adjusting it slightly to fit the new test case. This can result in duplication of code, so-called code clones, the result of which make the test suite bloated and harder to maintain. Eldh [5] found that generally 30%–50% of test cases in test suites have code cloning overlap, with some test suites measuring up to 80%. This problem of poorly designed test cases is often due to test code not receiving the same level of attention as the code used in production. Test cases often break with the principle of generality; instead of one general test case which can be used to cover many different scenarios, there are multiple test cases with minor variations between them.

For testing in Python, `pytest` [6], a powerful and flexible testing framework, has emerged as a popular choice among developers due to its simplicity and scalability for writing test code. This article focuses on reducing `pytest` test suites through parametrization. This is done by presenting and describing the implementation of a novel tool for refactoring test suites: Pytest Test Refactorer<sup>3</sup>, or PyTeRor for short.

<sup>3</sup> The source code for PyTeRor can be found at <https://github.com/semaki2000/PyTeRor> and is archived at <https://doi.org/10.5281/zenodo.11145543>; also see [8].

```

@pytest.mark.parametrize("input1, input2, expected",
                          [(1, 2, 3), (5, 10, 15)])
def test_calculator_add(input1, input2, expected):
    calculator = Calculator()
    result = calculator.add(input1, input2)
    assert result == expected

```

Fig. 1: Usage of `pytest` parametrization

*Code clones* is a term referring to fragments of source code that are equal or similar to each other. Code clones can occur through developers purposely copy-pasting code from one place to another. They can also occur inadvertently, for example by developers following the same specific patterns when writing code, such as the way they iterate through a list. Code clones are also referred to as duplicated clones or software clones in the literature.

A *code fragment*, or fragment of code, is a single continuous piece of code. When two fragments of code are clones of each other, we call it a *clone pair*. One fragment of code can be part of multiple different clone pairs. A *clone class* is a group of code fragments, where each fragment is in a clone pair relation with each other fragment in the group.

The threshold of similarity required for two fragments of code to be considered code clones is only vaguely defined in the literature. To combat this vagueness, syntactical clones are generally classified into three types (we elide a fourth type, semantic clones) [13]: *Type 1* or *Exact* clones, where two fragments of code are identical. This category allows for differences in the surrounding comments and whitespace, but when that is abstracted away, the fragments are exactly the same.

*Type 2* or *Renamed* clones are like Type 1 clones, except they can also have differences in literals, identifiers, and types. As none of these elements affect the structure of the code, Type 2 clones are structurally equal. Although actual values may be different, the structure of statements and the way the code is built up is identical between two Type 2 clones. They can be further distinguished between *consistent* or *blind* clones, depending on whether there exists a consistent renaming of identifiers between the members of a pair.

*Type 3* or *Gap-clones/Near Miss* clones are like Type 2 clones, except they can also have some statements added or removed. The accepted variation between two code fragments to classify them as Type 3 clones lacks a clear, consistent threshold, as different tools and literature have different definitions of what makes a Type 3 clone pair.

The code duplication that our approach eliminates are hence Type 2 clones, that is within method bodies, where values in function calls and expected results in assertions differ, but the overall structure of the body is the same. We achieve this by introducing parameters for the actual values used in the test, and using the parametrization-annotation of `pytest` to provide the single remaining copy of the test with multiple instances [6]. Figure 1 shows an example of a `pytest`

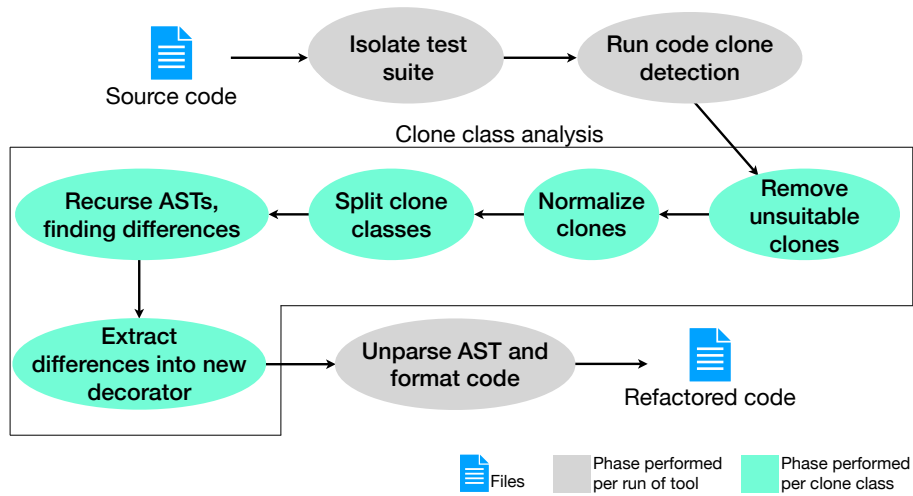


Fig. 2: Phases of PyTeRor

parametrized test. Note the correspondence between the string literals in the annotation and the formal parameters of the method.

This article is structured as follows: Section 2 gives a technical overview of our approach. In Section 3 we evaluate our implementation on open source repositories. Finally, in Section 4, we present an outlook and our conclusions.

## 2 Design

In this section we describe the architecture and workflow of our tool PyTeRor, which identifies suitable candidates and refactors them into parametrized tests. Figure 2 shows the different phases of PyTeRor.

*Clone detection.* To successfully apply PyTeRor, the target code base must obviously contain existing `pytest` tests. As there is no standard project-layout for Python projects, the test suite must be located in the project. We then invoke NiCad [4], an existing scalable, flexible code clone detector (top row in Figure 2). NiCad comes with an existing Python grammar and a configuration for Type 2 clone detection, to which we make some small adjustments: with the original grammar, the `None` literal is parsed as an identifier, which causes problems in the refactoring further down the line<sup>4</sup>. As for the configuration, the two key-configurations are that we reduce the minimum size of clones to just one line, and that we are interested in blind, not just consistent, clones, as this will allow us to parametrize both identifiers and literals, instead of just the latter. For a

<sup>4</sup> See our support request in <https://tx1.ca/forum/viewtopic.php?f=22&t=1148>.

more detailed discussion of the use of NiCad, we refer the reader to Kingston’s thesis [8].

The other two rows in Figure 2 describe the core functionality of PyTeRor: from NiCad’s set of detected clones (exported as a data structure via XML), we remove any clones that are not in the bodies of methods following `pytest`’s convention for test cases (method names prefixed with `test_` or not in a class whose name starts with `Test`). We also do not handle tests that already make use of non-trivial parametrization — this mostly excludes tests where parameters are computed in some way and more invasive structural updates would be necessary to enable reuse.

Next, we *normalize* the AST of all clones by removing additional decorators and docstrings. We keep track of them to recombine them later. This allows us for example to merge a test into an existing parametrized test by adding the required additional parameters to the existing decorator.

The detected clones also need to be *split into distinct clone classes*. Not every detected clone pair lends itself automatically to parametrization. The goal of these splits is to ensure that, upon refactoring, clones will be kept within their own scopes, so as to avoid capturing names. Therefore, after splitting clone classes based on scope, each of our new clone classes should contain exclusively clones in the same scope. The clone class is split here based on three rules that should guarantee this property:

1. There are clones in different modules.
2. One or more clones are in the global scope whilst one or more clones are inside a class-scope.
3. There are clones in different class-scopes.

If any of these rules are fulfilled, the clone class will be split. For clone classes spread out over  $N$  modules, the clone class is split into  $N$  new classes. Likewise, for class scopes, if a clone class is spread out over  $N$  classes, we split into  $N$  new clone classes. All clones in the global scope of the same module are split off into a separate class as well.

PyTeRor also provides some facilities to combine clones detected in distinct files, provided they are in the global scope, which we call *cross-file parametrization*. More ways of splitting the clone class, such as splitting based on attributes, are performed at a later point in the pipeline. These cannot be performed along with the initial splits on scope and decorators, as they are dependent on information gathered from the next step.

We adopt the *AST analysis* to compute the difference of clone pairs, and check if they are amenable to refactoring, using Python’s `ast` module. We recursively check that if AST node types match for each clone in the class. This phase can lead to additional splitting, but otherwise results in a list of differences in the `Constant`, `Name` and `Attribute` nodes occurring in the ASTs. As these are syntactical differences, we also need to check the scope of any variables — we cannot refactor a mix between local and non-local variables. The straightforward case is refactoring all-global variables, whereas some additional steps must

```

def test_a():
    a, b = 1, 2
    a + a

```

```

def test_something():
    something, other = "some", "text"
    something + something

```

Fig. 3: Consistent local variables

be taken for all-local variables. Figure 3 shows an example for two consistent, all-local clones that can easily be refactored.

Finally, we generate the refactored test with a `parametrize`-decorator, taking care to introduce fresh identifiers where necessary, combining any docstrings, and unparsing the AST. Figure 4 shows the before and after of two unit tests being refactored into a single parametrized unit test according to our scheme.

### 3 Evaluation

In this section, we describe how we evaluated our tool. This includes choosing available — ideally representative — sample input to refactor, and the exact metrics to evaluate against.

#### 3.1 Picking Code Repositories

As there is no pre-existing library of code bases containing a large amount of Type 2 clone test functions, we must create our own set of code bases to evaluate against. In doing this, there are multiple attributes we must take into account.

Firstly, and most importantly, we want repositories with `pytest` test suites. If there are no `pytest` tests in a suite, none of the tests can be parametrized. Repositories which do not recognizably utilize any specific testing framework can be picked, as we can import `pytest` in order to use parametrization when needed. This is only the case if the test suite follows the `pytest` naming conventions. Suites which use both `pytest` and another testing framework such as `unittest` can be part of the evaluation, as the `pytest` tests can still be parametrized. However, this will lead to problems if `PyTeRor` attempts to parametrize `unittest` tests.

Another attribute to take into account is that test suites must be of a certain size. This is because we are looking for test suites containing Type 2 code clones. As the size of the test suite grows, the potential clones grows with it. Not all clones can be parametrized, so a higher amount of clones is better. The arbitrary minimum threshold we use for the amount of defined tests a repository must contain to be relevant is 100 defined tests.

To find repositories, we use GitHub, as it is a go-to platform for open-source code. The process of finding candidate repositories is performed manually, by trawling GitHub, filtered by Python as a topic, looking into different repositories

<pre>def test_multiplication_simple():     calc = Calculator(precision=4,                       unit="deg")      a, b = 2, 3     expected = 6      actual = calc.multiply(a, b)     assert actual == expected</pre>	<pre>def test_multiplication_advanced():     calc = Calculator(precision=7,                       unit="rad")      a, b = 0.3145, 4.2535     expected = 1.3377258      result = calc.multiply(a, b)     assert result == expected</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

a) Two unit test code clones (Type 2)

```
@pytest.mark.parametrize(
    "parametrized_var_0, parametrized_var_1, parametrized_var_2,
    parametrized_var_3, parametrized_var_4",
    [
        pytest.param(4, "deg", 2, 3, 6, id="test_multiplication_simple"),
        pytest.param(7, "rad", 0.3145, 4.2535, 1.3377258,
                     id="test_multiplication_advanced"),
    ],
)
def test_multiplication_simple_parametrized(
    parametrized_var_0,
    parametrized_var_1,
    parametrized_var_2,
    parametrized_var_3,
    parametrized_var_4,
):
    calculator = Calculator(precision=parametrized_var_0,
                           unit=parametrized_var_1)

    (a, b) = (parametrized_var_2, parametrized_var_3)
    expected_result = parametrized_var_4
    actual_result = calculator.multiply(a, b)
    assert actual_result == expected_result
```

b) Code clones combined and refactored into a single parametrized unit test

Fig. 4: Unit tests pre- and post-refactoring

to investigate their test suites. If a repository has a sufficient number of tests, we check for the existence of code clones in the test suite by running NiCad on it. We also ignore repositories which have configured `pytest` in such a way that the standard test discovery rules are no longer valid. This is because PyTeRor does not support this configuration.

In order for a repository to qualify for the evaluation, installing requirements and running the test files should be achievable with relative ease. Repositories in

<i>repo#</i>	<i>URL &amp; commit hash</i>
1	<a href="https://github.com/onekey-sec/unblob.git">https://github.com/onekey-sec/unblob.git</a> f4df4165e2452833eaced99307547c72c1e6d4e9
2	<a href="https://github.com/pallets/flask.git">https://github.com/pallets/flask.git</a> b90a4f1f4a370e92054b9cc9db0efcb864f87ebe
3	<a href="https://github.com/psf/requests.git">https://github.com/psf/requests.git</a> a58d7f2ffb4d00b46dca2d70a3932a0b37e22fac
4	<a href="https://github.com/tiangolo/fastapi.git">https://github.com/tiangolo/fastapi.git</a> 1b105cb00dbf14157c38467ecc728447de49c8d
5	<a href="https://github.com/sphinx-doc/sphinx.git">https://github.com/sphinx-doc/sphinx.git</a> 768cf5e7ac63c7b26b6f3bd9cdfd2c5bf8ba4654
6	<a href="https://github.com/nvbn/thefuck.git">https://github.com/nvbn/thefuck.git</a> c7e7e1d884d3bb241ea6448f72a989434c2a35ec
7	<a href="https://github.com/davidhalter/parso.git">https://github.com/davidhalter/parso.git</a> 279fd6903ec9575f233067f3fc7b47f6fd6705d0
8	<a href="https://github.com/davidhalter/jedi.git">https://github.com/davidhalter/jedi.git</a> a4574a50d01e88316ddf554419cae64f547a7d70
9	<a href="https://github.com/pylint-dev/astroid.git">https://github.com/pylint-dev/astroid.git</a> 7a3b482b9673243d2ccc895672eb1e452f5daa82

Table 1: Repositories used in evaluation

which the tests cannot be run, for some reason or other, have not been included. To gather data on how many tests pass and fail before refactoring, we run the tests before we use PyTeRor on the repository. After running PyTeRor, we will again run the tests, then compare the results.

After considering these factors, we picked nine open source repositories for our evaluation (see Table 1). For the detailed steps necessary to run the test cases see our artefact and accompanying thesis [8]. We leave automated trawling for repositories and test case execution to future work.

### 3.2 Metric Selection

The goal of the evaluation is two-fold: firstly, we want to measure whether the refactoring itself was successful. This implies that we changed the structure of the code, without changing its functionality. Secondly, we want to measure whether we managed to reduce the size of the test suite. We want to do this both for standard PyTeRor, and for PyTeRor with cross-file parametrization enabled.

In order to measure whether the refactoring itself was a success, we can compare the running of the tests before and after utilising PyTeRor. Here, there are three key metrics: number of tests run, number of tests passed, and number of tests failed. Optimally, an equal amount of tests should be run, with the same numbers of passing and failing tests as before PyTeRor was employed on the

code base. If this is not the case, we have not refactored successfully. Should the number of tests in total be different, we will have inadvertently removed tests from the test suite. Should the number of passing and failing tests be different, we will have inadvertently changed the behaviour of certain tests. Even if the amount of passing and failing tests should be the same, the possibility still exists that some of the tests may have “switched” places. For example, one test which previously passed could now fail, whilst one test which previously failed could now pass. This would be an error in our refactoring, but would not be noticeable using this metric. The number of tests switching places can also be generalised:  $N$  previously passing tests may now fail, likewise  $N$  previously failing tests may now pass. Without performing a manual check of the `pytest` output, we have no way of asserting that it is the same tests passing and failing as previously.

We propose the following metric to evaluate if the test suite has been effectively reduced: it is important to separate between tests and test runs. A single test can have multiple test runs, due to the `parametrize` decorator. However, it is still only one test. When measuring the number of tests, we are referring to the number of actual `def test_` constructs in the code, and not parameterized instances of the test. These `def test_` constructs must be either in the global scope or in a class scope. In order to measure the number of tests, PyTeRor includes an option for measuring the amount of tests contained in the test suite pre-refactoring. This option also measures how many tests have been removed by PyTeRor due to parametrization. Both of these values are output, along with the amount of tests removed due to parametrization as a percentage of total pre-existing tests, when using the `-ex` or `-experiment` option.

We should also take into account how many potential clones existed in the first place. Therefore, we add the total number of tests removed and the total number of target clones successfully parametrized, then calculating this sum as a percentage of the total number of relevant clones in the suite. This number gives us a more accurate view on the amount of clones which will no longer be discoverable as clones, either due to their own removal from the suite, or due to the removal of other clones from the suite.

### 3.3 Evaluation

Tables 2 and 3 present the results of our evaluation and the relative measures. In Table 2, for each of the nine repositories ( $\#$ ), we show the number of contained *test definitions* ( $T$ ), and the number of ( $p$ )assing, ( $s$ )kipped or ( $f$ )ailed *test runs* as a baseline. The numbers in parentheses denote tests that *xfailed* or *xpassed*<sup>5</sup>. Tests which *xpassed* are in parentheses next to the number of passed tests, whilst tests which *xfailed* are next to the failed tests. The following columns show for both modes of operation ( $r$ egular and  $c$ ross-file parametrization) the number of detected clone classes ( $cs$ ), the number of clones ( $cl$ ) as identified by PyTeRor, the number of tests removed ( $tr$ ), and the number of clone classes which have been parametrized ( $cp$ ). The final batch of columns indicates changes in passing,

<sup>5</sup> *xpassed* tests are passing tests which were expected to fail (*xfail*).



#	$T$	$p$	$s$	$f$	$cs$		$cl$		$tr$		$cp$		$p/s/f$ ( $diff$ )	
					$r$	$c$	$r$	$c$	$r$	$c$	$r$	$c$	$r$	$c$
1	161	791	0	92	2	2	4	4	1	1	1	1	0/0/0	0/0/0
2	383	482	1	1	48	44	8	6	1	1	1	1	0/0/0	0/0/0
3	329	587	14	1	16	16	23	23	6	6	5	5	0/0/0	0/0/0
4	1908	2004	75	1	309	312	1197	1172	270	435	130	119	-5/0/+5	-250/0/+260
5	1367	2075	31	2	73	75	215	215	100	112	53	51	0/0/0	0/0/0
6	640	1887	0	0	84	83	371	143	2	29	2	11	0/0/0	-25/0/+25
7	282	1351	0	0	21	21	92	91	73	72	17	17	0/0/0	0/0/0
8	483	3863	9	11(5)	14	14	24	24	11	11	6	6	-2/0/+1	-2/0/+1
9	422	1635(1)	25	3(16)	23	22	14	12	4	4	4	4	-2/0/+1	-2/0/+1

Table 2: Evaluation results of (r)egular parametrization and (c)ross-file parametrization

#	$tests\ removed\ (\%)$		$clones\ removed\ (\%)$	
	$r$	$c$	$r$	$c$
1	0.62%		50.00%	
2	0.26%		25.00%	33.33%
3	1.82%		47.83%	
4	14.15%	22.80%	33.42%	45.82%
5	7.32%	8.20%	71.16%	75.81%
6	0.31%	4.53%	1.08%	11.05%
7	25.89%	25.53%	97.83%	97.80%
8	2.28%		70.83%	
9	0.95%		57.14%	66.67%

Table 3: Relative measure of the results of (r)egular parametrization and (c)ross-file parametrization

skipped or failed tests after applying the refactoring in the corresponding mode with respect to the baseline.

*Regular parametrization.* We observe the following interesting cases where we have affected the test-suite: in repository 4, five additional tests failed post-refactoring. These failing tests were invoking other tests. They failed because the tests they were invoking had been removed as part of the refactoring, leading to an error.

In repositories 8 and 9, two fewer tests passed, but only one more test failed. These interesting cases of the one disappearing test are due to PyTeRor parametrizing two tests within a `unittest` test class. `pytest` can run tests written using the `unittest` framework. However, `pytest` runs these tests as

`unittest` tests, not as `pytest` tests. Therefore, when we attempt to parametrize these tests using `pytest`'s own `parametrize` decorator, the decorator is not recognized, leading the test to fail. Another consequence of not recognizing the decorator is that the parametrized test is seen as a single test item, rather than multiple.

*Cross-file parametrization.* In repositories 1, 2, 3, 8 & 9, the same number of tests were removed as using the default “regular” option. For repositories 2, 7 & 9, fewer clones and clone classes were found than using the default option, but this did not affect the results, apart from the percentage of clones removed, shown in Table 3, which naturally increased. Both repository 1 and 3 had identical results between the default and the cross-file option.

Repository 5, where 12 additional tests were removed, despite fewer clone classes being found than with the default option. These removed tests did not affect the number of passing and failing test items. The 12 additional tests that were removed would have been identified using the default option as well, but would have then been discarded as they contained differences in scope. As we discovered the same amount of relevant clones, but removed a slightly higher amount of tests, the percentages across files are slightly higher than the corresponding percentages for regular parametrization of repository 5, shown in Table 3.

However, the refactoring was *unsuccessful* for the following repositories: in repository 4, 165 additional tests were removed. 245 fewer test items passed compared to the default option (and likewise, 255 additional test items failed). The discrepancy between additional tests removed and additional test items failed is due to the fact that many of the 165 tests that were removed contained multiple test items, due to being parametrized.

In repository 6, 27 additional tests were removed. 25 fewer test items passed compared to the default option (and likewise, 25 additional test items failed). The large majority of cross-file parametrizations in this repository lead to failed tests, and thereby also failed refactorings.

### 3.4 Discussion

Apart from these two exceptions which constitute seven failed tests across all test suites in the evaluation, the *refactoring was successful*. In total, 468 tests were removed from the suites, though over half of these were from a single repository (270, repository 4). Many of the discovered clones were unparametrizable for some reason or other. Eldh found that there was typically a 30%–50% code cloning overlap in test suites [5], though this was generally for larger test suites than we are looking at, and accounted for clones of types 1–3. Despite only looking for Type 2 clones, for some repositories, we found similar numbers of clones. These were mainly repositories 4, 6 & 7, of which 4 & 7 had large numbers of *parametrizable* clones. Generally, the number of clones as a percentage of the number of tests in total was highest for the largest test suites. This means a higher share of tests in the largest test suites were clones than in the smaller

test suites, the one exception being repository 7, with less than 300 tests, and almost 100 clones. Extrapolating the general trend, had we chosen repositories with even larger test suites, we could perhaps have found an even higher percentage of clones, as found by Eldh. However, a high number of clones does not necessitate a high number of *parametrizable* clones.

Although the refactoring was successful, for most of the repositories the test suites themselves were *not reduced in size by large amounts*. For four of the repositories, less than 1% of tests were removed, with two more repositories hovering around 2%. However, the low number of removed tests in these repositories can be explained by the low number of clones found in them, with the exception of repository 6, which consists of a large number of clone classes spread out across multiple files. For the three remaining repositories in the evaluation, the test suites were reduced significantly, ranging from a 7% to a 26% reduction in the number of tests. For these repositories, with this clear reduction in the number of code clones, we argue that the refactored test suites have improved maintainability when compared to the same test suites pre-refactoring. This is because as the number of code clones in the test suite decreases, maintainability of the suite increases. Rather than having to maintain multiple similar tests which are clones, a single test can instead be maintained.

Our *cross-file parametrization does not work in most cases*. For it to work in the same way we have parametrized other clone classes, it requires all clones in a class to have the same non-local variables, containing the same values, despite the fact that the clones exist in different scopes. For our cross-file implementation, we failed to take into account that same names in different scopes could have different values. Asserting that the same names across different scopes have the same values can be a difficult task, and using static analysis, is in many cases impossible, as we cannot know the runtime value. If we assume that the assertion (same name, same value) holds true for a clone class, this effectively means that the clones are exactly the same test, except for potential differing literal values. Thus, they can be parametrized. Of the cross-file clone classes we have anecdotally looked at, the large majority have differences in non-local names. In this way, they cannot be parametrized using our current solution.

*Multiple facets of PyTeRor affect code quality negatively*. The generated names are always of the form `parametrized_constant_X` and `parametrized_name_X`, where X is a number. Generally, good variable names describe the value they contain in some way. These generated variables names do not. In fact, by extracting and replacing the original global variable names in the target clone with generated names, we are reducing the quality of the code by making it more difficult to understand. However, these situations where variable names are extracted and replaced with a new generated variable name only occur when clones use different global variables in the same place. In order for the global names to be parametrized, they *must* be extracted into the parametrize decorator and represented through another variable name. Optimally, the new variable name should have something to do with the extracted names, so that it is easier to understand what is going on inside the function. The form `parametrized_name_X`

does not build much understanding of what the variable is used for. However, it does immediately communicate that the variable is parametrized, leading the reader to the parametrize decorator in order to see what the actual value is for each test run. Removal of comments effects understandability of the refactored code negatively.

Though all this obfuscation is to the detriment of code quality, one could also argue that *removing clones from the test suite improves the code quality*. The code becomes easier to maintain in line with the usual argument for clone elimination: previously, changes which were required for all clones in a class had to be made independently for each clone. Post-refactoring, these changes only need to be made in a single place, the target clone. In addition to practicality, this can be helpful for multiple reasons, e.g., removing bugs or vulnerabilities. When clones in a clone class share a common bug or vulnerability, by parametrizing and removing these clones, the bug or vulnerability only needs to be patched in a single place, rather than multiple.

### 3.5 Threats to validity

In this section we will account for some threats to the validity of the evaluation on whether our tool can successfully and correctly replace clones through parametrization.

*Reliability.* We are entirely dependent on our clone detector, NiCad, to identify and inform us on which functions are clones. Except for cross-comparing all tests in a suite ourselves, which quickly becomes infeasible as the test suite grows, we have no way of asserting that we have discovered all clones in the test suite. NiCad itself has been shown previously to have some bugs in its Python grammar, and there could plausibly be more. For example, it seems not to identify clones with slight differences in formatting, such as the example in Figure 5 (line breaks added for presentation), which is from Repository 3. The first two of these clones are correctly identified, and therefore also parametrized. However, the third, `test_should_strip_auth_http_downgrade`, is not considered by NiCad to be a clone of the others, and is therefore not parametrized by PyTeRor.

*Internal validity.* Unfamiliarity with the projects themselves and the test suites they contain can also be a threat to the validity of the evaluation. This can be a problem on multiple fronts. Firstly, we do not have exhaustive knowledge of the code we are running the evaluation against. This creates problems such as those identified above, where we cannot assert whether we know of all clones in each test suite. Secondly, we could be missing some important characteristics about the test suites. For a concrete example we look to Repository 4, where many of the discovered clones are within the directory ‘`test_tutorial`’, which may in explain to a certain extent why over half of the tests in the test suite of Repository 4 are clones. Without familiarity of the projects we are testing on, we lack knowledge of specific characteristics which could be affecting the results of the evaluation.

```

def test_should_strip_auth_port_change(self):
    s = requests.Session()
    assert s.should_strip_auth(
        "http://example.com:1234/foo", "https://example.com:4321/bar"
    )

def test_should_strip_auth_host_change(self):
    s = requests.Session()
    assert s.should_strip_auth(
        "http://example.com/foo", "http://another.example.com/"
    )

def test_should_strip_auth_http_downgrade(self):
    s = requests.Session()
    assert s.should_strip_auth("https://example.com/foo",
                               "http://example.com/bar")

```

Fig. 5: Example of clones not identified by NiCad

*External validity.* Another point which can strongly affect the results of the evaluation is the choice of repositories. Multiple characteristics were required for a repository to be included in the evaluation. The repositories were required to be open source, contain at least 100 tests and be easily runnable using `pytest`. Each of these requirements may have affected the outcomes of the evaluation. For instance, a closed-source repository may be less maintained, and could therefore contain a higher number of clones in its test suite than a comparable open-source repository. Likewise, the same proposed correlation could exist between the ease of running a test suite and the number of code clones it contains. Thereby, we could be introducing biases into the evaluation with the repositories we are choosing to include. With regards to open-source vs closed-source repository, Raghunathan et al. found no difference in software quality between the two development practices [12]. Furthermore, Monden et al. mention the correlation between software quality and the existence of code clones in a project [11]. However, the two definitions of software quality differ somewhat, meaning we cannot extrapolate that comparable closed-source and open-source versions of software would contain a similar amount of code clones. Similarly, we have not considered the development processes for the selected repositories.

Most importantly for the results of the evaluation, the current number of repositories included in the evaluation is relatively small. We do not have enough repositories to speak confidently of the usefulness of PyTeRor in removing code clones from test suites, and the current choice of repositories shows quite some variance. If all repositories displayed relatively similar results, in terms of similar percentages of tests removed from their test suites, we could perhaps point in a general direction with regards to the effectiveness of PyTeRor. Such varying results, however, leave us far from any conclusion of significance.

*Construct validity.* As we change the names of tests, we cannot exactly map test runs before and after to each other. Hence we do not know whether the passing and failing tests are the same pre-refactoring as post-refactoring; we only know that the totals are the same. Additionally, when counting removed clones, we include target clones which may in fact still be a clone of other tests in the suite, either due to splitting of the clone class or singular clones being discarded from the parametrization of the class.

## 4 Conclusion

Through PyTeRor, we have demonstrated that it is possible to reduce the size of `pytest` test suites by automatically refactoring Type 2 code clones using parametrization. We have delineated the implementation of PyTeRor in some detail, bringing up and discussing issues along on the way. In designing and conducting an evaluation, we have performed this refactoring on nine open source repositories from Github.com. Evaluating the results, for some of these repositories we found and parametrized large numbers of clones, whilst for others we found few clones, though all repositories had at least one test removed. We discussed the results of the evaluation and other aspects of PyTeRor. These aspects were the quality of the refactored code, and whether it was a true refactoring, due to the fact that certain behaviour is not preserved after applying `pytest` on the refactored code.

Through our evaluation, we have demonstrated that PyTeRor has the ability to parametrize Type 2 code clones in `pytest` test suites, though there are also some Type 2 clones which PyTeRor cannot parametrize, such as those with attribute differences or differences in scope. Though there are some exceptions, the refactored tests are generally correct, producing the same behaviour as pre-refactoring. We argue that this refactoring increases the maintainability of `pytest` test suites, as we reduce the number of code clones in a suite.

### 4.1 Related Work

Though we have not found other work performing the same kind of automated refactoring of code clones in test suites for other languages or frameworks, much work has been done relating to multiple aspects of this thesis.

Zhang et al. created a tool for performing fully automated refactorings of Python code. However, unlike PyTeRor, this tool targets non-idiomatic Python code, with the intention of transforming it into idiomatic Python code. This is done for nine specific idioms identified by Zhang et al. [18].

Xuan et al. performed automatic test code refactoring [17], though this refactoring is performed with the intention of improving dynamic analysis, rather than to reduce the number of code clones in a test suite, as has been our goal. The concept of test code refactoring in general has also been covered by Deursen et al. [16], displaying eleven different test smells, as well as six refactorings for

these. Manual test refactorings have also been discussed in detail by Meszaros, in his book *xUnit Test Patterns: Refactoring Test Code* [10].

With regards to refactoring detected clones, some previous work exists. Baars and Oprescu target clones in object-oriented languages, detecting clones which are refactorable [1]. However, they do not target clones in test suites specifically, nor do they introduce an automated refactoring solution.

Refactoring code clones in test suites has been covered by Tsantalis et al. [15], though test code was not the main focus of their work. They found that clones in production code were more often refactorable than clones in test code. They found that clones with close relative locations are more refactorable than clones in distant relative locations. This relates to our findings that cross-file `pytest` code clones often are not refactorable. Additionally, they found that NiCad’s consistent renaming option is most suitable for detecting code clones meant for refactoring. In practice, this may be true, as clones must have consistent local names to be candidates for our refactoring. However, through the implementation of PyTeRor, we found that this option may miss certain types of clones, namely clones with consistent local names but inconsistent non-local names. In order to detect as many refactorable clones as possible, blind renaming in NiCad, in combination with filtering out clones with inconsistent local names, allows detection of potentially higher numbers of refactorable clones.

Work on automated clone detection and elimination has also been published for other languages, examples being Erlang [9] and Haskell [3]. Numerous other techniques and tools for semi- or fully automated refactoring of code clones also exist, as shown by the literature review conducted by Baqais and Alshayeb [2].

## 4.2 Future Work

*Parametrizing test clones in other languages.* One interesting topic of research is performing this same parametrization-refactoring for testing frameworks in other languages. An example of such a framework is JUnit, a testing framework for Java, which can parametrize tests through the `@ParameterizedTest` annotation (as of JUnit 5). For single parametrized values, the `@ValueSource` annotation can be used. For multiple values, either the `@CsvSource` annotation, which takes multiple strings containing comma separated values, or the `@MethodSource` annotation, which takes a string containing method name, can be used.

Many xUnit-style testing frameworks have features allowing parametrization of tests, such as xUnit.net (C#/.NET) and cppUnit (C++). All frameworks with such features would be candidates for parametrization-refactoring as done in this paper.

*Qualitative evaluation.* The choice of clones to consider and the exact form in which to refactor them (e.g., choosing the necessary fresh names for parameters) deserves a closer look. Clones are not universally seen as bad [7], and developers may have preferences for what or how to refactor. Developer feedback would be valuable input before taking our work further.

*Larger-scale evaluation.* Evaluating the effectiveness of PyTeRor in parametrizing code clones would be made easier if we could benchmark results against a standard or database of `pytest` code clones. Benchmarks like this are used to evaluate metrics of tools in related fields, one such example being BigCloneEval [14], which can be used to evaluate the effectiveness of code clone detectors. Another interesting continuation of the research from this thesis would be to run the same evaluation on a larger scale. By creating a script which could clone git repositories, run `pytest` and collect results, run PyTeRor, then run `pytest` again, comparing results to the first run, we could automate the evaluation to be run on a large number of test suites. This would give us more data, which again could lead to interesting analysis, both on the effectiveness of PyTeRor and on the state of `pytest` test suites in general, with regards to Type 2 code clones. Of course, we would still only be able to analyse open-source test suites, and any conclusion made would not necessarily hold true for closed source test suites.

*Continuing work on PyTeRor.* In terms of continuing the work on PyTeRor itself, handling `pytest`'s many options for init files would go a way to making PyTeRor universally applicable on `pytest` test suites, including those overriding the default test discovery rules. Additionally, extraction of common initialisation code into fixtures is also an interesting area of research. This method of refactoring could potentially reduce the size of many tests within the test suite, in addition to improving code quality and maintainability. This could be implemented either as an extension of PyTeRor, or alternatively as a separate tool.

*Integration into IDEs.* We see some potential for integrating clone-detection and our refactored solution as a suggestion into integrated development environments (IDEs). Clones in tests, just like regular clones, can surreptitiously creep into a code-base, but will most likely be more clustered in the project structure than general clones. We conjecture that novice beginners might have some benefit from being introduced to the concept of test case parametrization through interactive use, and hence subsequently make direct “proper” use of the testing framework.

## References

1. Baars, S., Oprescu, A.: Towards automated refactoring of code clones in object-oriented programming languages. In: Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE 2019). CEUR Workshop Proceedings, vol. 2510. CEUR-WS.org (2019)
2. Baqais, A.A.B., Alshayeb, M.: Automatic software refactoring: a systematic literature review. *Software Quality Journal* **28**, 459–502 (2020). <https://doi.org/10.1007/s11219-019-09477-y>
3. Brown, C., Thompson, S.: Clone detection and elimination for Haskell. In: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. p. 111–120. PEPM '10, ACM, New York, NY, USA (2010). <https://doi.org/10.1145/1706356.1706378>



4. Cordy, J.R., Roy, C.K.: The NiCad clone detector. In: 2011 IEEE 19th Intl. Conf. on Program Comprehension. pp. 219–220. IEEE (2011). <https://doi.org/10.1109/ICPC.2011.26>
5. Eldh, S.: On technical debt in software testing - observations from industry. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering. LNCS, vol. 13702, pp. 301–323. Springer (2022)
6. Holger Krekel and others: `pytest`: helps you write better programs (2024), <https://docs.pytest.org/en/8.2.x/>
7. Kapser, C., Godfrey, M.W.: "Cloning Considered Harmful" considered harmful. In: 13th Working Conference on Reverse Engineering. pp. 19–28. IEEE (2006). <https://doi.org/10.1109/WCRE.2006.1>
8. Kingston, S.: Automated Clone Elimination in Python Tests. Master's thesis, University of Oslo (June 2024), <https://www.mn.uio.no/ifi/english/research/groups/psy/completedmasters/2024/automated-clone-elimination-in-python-tests.html>
9. Li, H., Thompson, S.: Incremental clone detection and elimination for Erlang programs. In: Giannakopoulou, D., Orejas, F. (eds.) Fundamental Approaches to Software Engineering. LNCS, vol. 6603, pp. 356–370. Springer (2011)
10. Meszaros, G.: xUnit Test Patterns: Refactoring Test Code. Addison-Wesley Professional (2007)
11. Monden, A., Nakae, D., Kamiya, T., Sato, S., Matsumoto, K.: Software quality analysis by code clones in industrial legacy software. In: Proc. Eighth IEEE Symp. on Software Metrics. pp. 87–94 (2002). <https://doi.org/10.1109/METRIC.2002.1011328>
12. Raghunathan, S., Prasad, A., Mishra, B., Chang, H.: Open source versus closed source: software quality in monopoly and competitive markets. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans **35**(6), 903–918 (2005). <https://doi.org/10.1109/TSMCA.2005.853493>
13. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Queen's School of computing TR **541**(115) (2007), <https://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf>
14. Svajlenko, J., Roy, C.K.: BigCloneEval (December 2023), <https://github.com/jeffsvajlenko/BigCloneEval>, commit hash 6d393ec
15. Tsantalis, N., Mazinanian, D., Krishnan, G.P.: Assessing the refactorability of software clones. IEEE Transactions on Software Engineering **41**(11), 1055–1090 (2015). <https://doi.org/10.1109/TSE.2015.2448531>
16. Van Deursen, A., Moonen, L., Van Den Bergh, A., Kok, G.: Refactoring test code. In: Proc. 2nd Intl. Conf. on Extreme Programming and Flexible Processes in Software Engineering (XP2001) (2001), <https://ir.cwi.nl/pub/4324/04324D.pdf>
17. Xuan, J., Cornu, B., Martinez, M., Baudry, B., Seinturier, L., Monperus, M.: B-refactoring: Automatic test code refactoring to improve dynamic analysis. Information and Software Technology **76**, 65–80 (2016). <https://doi.org/10.1016/j.infsof.2016.04.016>
18. Zhang, Z., Xing, Z., Xia, X., Xu, X., Zhu, L.: Making Python code idiomatic by automatic refactoring non-idiomatic Python code with pythonic idioms. In: Roychoudhury, A., Cadar, C., Kim, M. (eds.) Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering, (ESEC/FSE). pp. 696–708. ACM (2022). <https://doi.org/10.1145/3540250.3549143>