Clone Detection in Test Code: An Empirical Evaluation

Brent van Bladel
University of Antwerp, Belgium

Serge Demeyer
University of Antwerp, Belgium
Flanders Make vzw, Belgium

***Abstract*—Duplicated test code (a.k.a. test code clones) has a negative impact on test comprehension and maintenance. Moreover, the typical structure of unit test code induces structural similarity, increasing the amount of duplication. Yet, most research on software clones and clone detection tools is focused on production code, often ignoring test code. In this paper we fill this gap by comparing four different clone detection tools (NiCad, CPD, iClones, TCORE) against the test code of three open-source projects. Our analysis confirms the prevalence of test code clones, as we observed between 23% and 29% test code duplication. We also show that most of the tools suffer from false negatives (NiCad = 83%, CPD = 84%, iClones = 21%, TCORE = 65%), which leaves ample room for improvement. These results indicate that further research on test clone detection is warranted.**

***Index Terms*—software clones; unit-tests; code clone detection**

I. Introduction

The recent popularity of agile software development has increased the emphasis on software testing for developers. In particular, test-driven development [1], and continuous integration [2], [3] require an effective test suite, which is executed early and often [4]. With each increment of the production code, the test code needs to be updated, extended, and maintained as well. Therefore, it is a recommended practice to continuously monitor the quality of the test suite [5], [6].

However, as agile teams aim to fix bugs and cover new features with the test suite, less time is spent on maintaining or refactoring the test code. This gives rise to the concept of "test smells": sub-optimal design choices in the implementation of test code [7], [8]. Duplicate tests (a.k.a. test clones) are one of the common symptoms, as the quickest way for a developer to test a new feature is to copy, paste, and modify an existing test [9]. Even if the developer does create a new test from scratch, the consistent structure of unit test code (the *setup-stimulate-verify-teardown* (S-S-V-T) cycle [10]) can still cause clones accidentally. A recent case study on a large project from industry found that 49% of the entire test code are clones [11]. This is significantly higher than the average of 7% to 23% for production code [12].

This high amount of duplicated test code can be problematic, as test smells (such as test code duplication) have been shown to have a strong and negative impact on program comprehension and maintenance [13]. Yet, research on test code duplication is limited, as most code cloning research focuses on production code.

In this paper, we perform an exploratory study on duplicated test code by running four clone detection tools (NiCad, CPD, iClones, and TCORE) on three open-source test suites. We classify and analyse a total of 2,544 detected clones, and compare the effectiveness of these tools. As such, we make the following contributions:

1) We show that test code in our dataset contains 23% to 29% duplication, which is higher than the average of 7% to 23% for production code.
2) We provide anecdotal evidence that clones in test code inherently differ from clones in production code.
3) We demonstrate that code clone detection tools suffer from many false negatives when run on test code, indicating that further research towards clone detection on test code is warranted.
4) We provide insights on test code clone detection in the form of lessons learned, in which we provide actionable advice on how to improve clone detection on test code.
5) Our dataset is publicly available, serving as a clone benchmark for test code.

The remainder of this paper is organised as follows. Section II provides the required theoretical background while Section III lists the related work. Section IV describes the experimental set-up, which naturally leads to Section V reporting the results. Section VI enumerates the threats to validity, and Section VII concludes the paper.

II. Background

Code clone. When two fragments of code from the source code are either exactly the same or similar to each other, we call them a code clone. A code clone is also synonymous with a software clone or duplicated code, as these terms can be used interchangeably.

Clone fragment. A fragment of code that is duplicated is called a clone fragment. Therefore, a code clone consists of two or more such clone fragments. When we consider a code clone that consists of exactly two clone fragments, we use the term clone pair. Most code clone detection tools report their results in terms of clone pairs.

Clone class. When a clone fragment is duplicated more than two times, we get a set of clone fragments called a clone class. Note that each combination of clone fragments in this set will also form a clone pair. One way to visualize the differences between these terms is to consider a graph: if every clone fragment is a node in a graph, then every edge between

two nodes is a clone pair, and a fully connected graph is a clone class. A clone class therefore consists of a set of clone fragments that all form clone pairs between themselves.

Clone types (1, 2, 3, 4). Code clones can be differentiated based on their degree of similarity. First, code clones can be divided into syntactic clones and semantic clones. Syntactic clones are code clones that are syntactically similar, and are further divided in three types: type-1, type-2, and type-3 clones. Type-1 clones are exactly the same, only allowing differences in comments, whitespaces, and indentation. Type-2 clones are a little less strict than type-1 clones as they also allow differences in variable names and literal values. Finally, type-3 clones are even less strict than type-2 clones. They also allow for lines of code in the clone fragment to be added or removed. Note that it is not required for these types of clones to be functionally similar, although the syntactic similarity usually does result in similar semantics (e.g. both clone fragments of a type-1 clone are functionally identical). Semantic clones on the other hand are code clones that are semantically similar without necessarily being syntactically similar, and are often called type-4 clones.

III. Related Work

A lot of research has already been performed on software clones. In 2007, Koschke performed a survey on the literature on software clones [12], which was followed in 2009 by him and his colleagues (Roy et al.) with an extensive comparison and evaluation of all code clone detection techniques and tools [14]. Since then, a lot of research has been performed to further investigate the prevalence, characteristics, impact, and detection methods of software clones. However, most of this research focuses on production code [12], [14], [15].

In 2012, Bavota et al. performed two empirical studies towards the effects of test smells, including test code duplication. Their results show that most test smells have a strong negative impact on the comprehensibility and maintainability of both the test code and the production code [13].

In 2015, Tsantalis et al. performed a large-scale empirical study using nine open-source projects. For their analysis, they used four different clone detection tools: CCFinder, Deckard, CloneDR, and NiCad. The focus of their study was on the refactorability of code clones in general, not specifically on test code duplication. However, they did briefly look at the difference between clones in test code and clones in production code. They found that in general test code contained more code clones than production code [16].

More recently, in 2018, Hasanain et al. performed an industrial case study that aims at better understanding code clones in test code. They used NiCad to detect clones on a large test suite provided by Ericsson. They found that 49% (in terms of LOC) of the entire test code are clones [11].

In this work, we build further on the research performed by Hasanain et al. [11]. We extend their work by investigating a different dataset of open-source test code using additional clone detection tools.

IV. Experimental Setup

In this section we provide a detailed description of the process we followed to reach our results. First we go over the tools and data we used, followed by the steps we took to perform our comparison.

A. Clone Detection Tools

There are many different code clone detection tools available, which can be divided in a few approaches. The three most common ones are (i) *text-based*, (ii) *token-based*, and (iii) *tree-based*. (i) Text-based approaches use the raw source code for comparison in the clone detection process, sometimes with a minimal amount of normalization (such as removal of empty lines and extra whitespaces). (ii) Token-based approaches begin by transforming the source code into a sequence of lexical tokens, which is then scanned for duplicated subsequences of tokens. (iii) Tree-based approaches use a parser to convert the source code into abstract syntax trees, which can then be scanned for duplicated subtrees using tree matching algorithms [14].

Clone detection techniques that do not fall under one of these three approaches have been proposed as well. For example, it has been shown that program dependency graphs (PDGs) and program slicing can be used to detect code clones [17], [18]. Other techniques include static analysis of memory states at each procedure exit point [19], or applying random testing to detect similar function output [20]. However, the latter two techniques cannot be applied to test code.

In order to select the tools for our comparison, we used the following criteria:

- *Availability*: To allow for our comparison to be easily reproduced, we selected tools which are publicly available for download. For example, *CloneDR* [21] was considered, but since this tool is not publicly available, we decided not to include it in our study.
- *Configuration*: To allow an accurate comparison between tools, we selected tools that are easily configurable in a similar manner (see Section IV-C). For example, *Deckard* [22] was considered, but we were unable to run it succesfully with the desired configuration.
- *Output*: To allow an automatic analysis of the results, we selected tools that have a structured output format. For example, *CCFinder* [23] was considered, but its output was not easily converted to our reference format (see Section IV-D).
- *Approach*: To allow for a more broad analysis, we selected tools with different approaches: one text-based, one token-based, one token- and tree-based hybrid, and one tree- and PDG-based hybrid.

Using these criteria, we selected the following clone detection tools:

- *NiCad* uses a text-based approach that performs clone detection in 3 stages. First it splits the input source into fragments of a certain granularity (e.g. blocks, functions). It then normalizes these fragments to a standard textual

TABLE I
DATASET DESCRIPTIVE STATISTICS.

Name	Files	Tests	LOC
Apache Common's Math	360	2,782	32,483
Google Guave	109	1,229	10,929
Java Design Patterns	133	313	3,747

This table only lists the JUnit test code of the respective projects.

form. Finally, the normalized fragments are linewise compared using an optimized longest common subsequence algorithm to detect clones [24], [25].

- *PMD's CPD* adopts a token-based approach based on the Karp–Rabin string matching algorithm on a frequency table of tokens in order to detect clones [14].
- *iClones* uses a token- and tree-based hybrid approach. First, it generates the abstract syntax tree of the source code and serializes it into a token sequence. Then it applies a suffix tree detection algorithm on this sequence in order to find clones [26], [27].
- *TCORE* uses a tree- and PDG-based hybrid approach, which focuses specifically on test code. First, it generates the abstract syntax tree of the source code. Then, it uses symbolic execution to create a PDG-like tree for each assert statement. Finally, it uses a tree matching algorithm on these trees to find clones [28].

B. Dataset

For our comparison, we selected three open-source Java projects: the Apache Commons Math library (from now on referred to as Apache), the Google Guava library (Google), and the Java Design Patterns library (Patterns). These projects were selected because they are popular and commonly used open-source Java projects with extensive test suites. All three projects make use of a continuous integration (CI) server that runs the test suite after each commit. At the time of analysis, all projects pass their CI build.

We use the test suite of these projects as the dataset for our comparison. Table I shows an overview of the test suite sizes of each project in terms of files, test cases, and lines of code (LOC). Note that the LOC metric does not include comments or blank lines. The Apache dataset is the largest of the three with 32k LOC, the Patterns dataset is the smallest with 3k LOC. We selected the projects specifically to have this difference in size to allow for more generalized results. The Google dataset, with 10k LOC, serves as a middleground.

C. Clone Detection

The configuration of a clone detector can have a large impact on the number and quality of clones detected by the tool. For each tool we opt for the default configuration for most parameters, as we assume that the default configuration would be best suited for a general purpose. There are only three parameters which we change: granularity, minimum clone length, and the output format.

In this research, we use a function level granularity, meaning that each clone fragment will consist of a function containing the cloned code. This makes it easier to match the same clone detected by multiple tools, since the start and end of the clone is then strictly defined by the start and end of the function. This has the added benefit that a cloned function corresponds to a cloned JUnit test case. NiCad has the option to select the granularity, which we set to function level. Both iClones and CPD default to a statement level granularity, which cannot be changed. TCORE defaults to an assert level granularity, which cannot be changed either. Therefore, we expand the clones detected by these tools to a function level granularity during postprocessing (see Section IV-D).

Because the size of a test can be significantly smaller than the size of functions in production code, and since we detect clones at a function (or test) level granularity, we choose to decrease the minimum clone length. By default, this minimum length is set to 10 lines of code for NiCad or 100 tokens for iClones and CPD. In our previous research, we found that half of the default (5 LOC or 50 tokens) is the best option for code clone detection in test code, as this allows for the smaller duplicated tests to be detected without generating many false positives [28]. Therefore, we set the minimum clone size parameter for NiCad, iClones, and CPD to half their default. Since TCORE detects duplicated assert statements, there is no minimum size to be set.

All four tools have the option to export the detected clones to an XML file. We choose this option as the structured XML output allows for easy and automated handling of the data. However, because there are differences in the XML structure used by the different tools, we perform a post-processing step to unify them into one reference XML format (see Section IV-D).

D. Postprocessing

After running the four clone detection tools on the three datasets, we have a set of 12 XML files with clones. To allow for easy analysis, we unify the XML files for each dataset into one file. These unified files are structured according to our reference format, as shown in Figure 1. As a result, we have an XML file per dataset containing all detected clones. To achieve this, we created a Python script that performs a series of postprocessing steps. Since the output differs between tools, these postprocessing steps are different for each tool.

NiCad's XML output is the closest to the reference format. It contains an XML element for each clone pair, with two child elements for the two fragments of the clone pair. These child elements contain attributes for the file, startline, and endline of the fragment. Since we configured NiCad to detect clones on a function level granularity, the startline and endline correspond to the start and end of the function containing the clone. The only postprocessing step we had to perform was removing extra elements and attributes (such as systeminfo, id, ...).

CPD's XML output differs significantly from the reference format. It contains the actual source code of the clone fragments, which causes syntax errors in the XML (for example when the $<$ operator appears in a fragment). Therefore, in the first step of postprocessing, we remove all the code fragments. We also rename the elements and attributes to match those of the reference XML format. Since CPD detects clone classes,

```
<clone type="T2" iclones="False" pmd="False" nicad="True" tcore="True">
  <source file="JavaDesignPatterns/FileSelectorPresenterTest.java" startline="95" endline="104"></source>
  <source file="JavaDesignPatterns/FileSelectorPresenterTest.java" startline="110" endline="119"></source>
</clone>
```

Fig. 1. Example of a clone in the reference XML format.

we expand these to clone pairs. This can be easily done by generating all combinations of clone fragments within each class. The next step is to expand the clones to a function level, since CPD does not consider function boundaries. Clone fragments that appear inside functions can be simply expanded up to the corresponding function by changing the startline / endline to match the start / end of the function. Clone fragments that span multiple functions need to be split into multiple function clones. Finally, we filter out all clone pairs that contain a fragment smaller than 5 lines of code.

iClones's postprocessing is the same as that of CPD, since it also detects clone classes and does not consider function boundaries. We start by renaming the elements and attributes. Next, we expand the detected clone classes to clone pairs. Then, we expand the clones to a function level in the same way as CPD's postprocessing. Finally, we filter out all clone pairs that contain a fragment smaller than 5 lines of code.

TCORE's XML output is similar to that of NiCad, and thus does not require much change either. However, because TCORE detects clones on an assert level granularity, we have to perform some postprocessing steps to expand the detected clones to function clones. First, we filter out all duplicate clone pairs. These are caused when an assert statement is located within a loop, since TCORE symbolic execution iterates the loop and encounters the same assert statement multiple times. Then, we filter out all clone pairs that occur within one function (i.e. test). These are caused when a test contains multiple similar assert statements. The next step is to expand the clones to a function level by changing the startline / endline to match the start / end of the test containing the assert. Finally, since TCORE does not have a minimum size parameter, we filter out all clone pairs that contain a fragment smaller than 5 lines of code.

After transforming all XML files to the same format, we merge them into one. Since all XML files now have the same structure and all clones the same granularity, we can easily detect where tools overlap and where they differ. We add four boolean attributes to each clone, one for each tool indicating whether or not the tool was able to detect the clone. Figure 1 shows an example of a clone in our reference XML format. Note that the type attribute seen in the example is not added yet during postprocessing, but after classification (see Section IV-E).

E. Classification

After postprocessing, we performed type classification on all detected clones. This classification was performed manually by the first author, and reviewed afterwards for correctness. In order to guarantee consistent classification and to remove any room for interpretation, a specific set of rules was followed to determine the type of the clone. Since we expanded all clones to a function level, these rules apply to matched functions / tests. In case multiple rules are satisfied, the rule that applies to the most lines of code is chosen. For example, a type-3 clone can contain multiple type-1 and type-2 clones. Also note that, since a type-4 clone considers the functionallity of the entire test case, it will always apply to the most lines of code, even if the entire body of the test case is a type-3 clone.

- *Type 1*: Both tests contain a continuous sequence of at least 5 lines of code that are exactly the same, not considering differences in comments, whitespaces, and indentation.
- *Type 2*: Both tests contain a continuous sequence of at least 5 lines of code that only differ in variable names and literal values, not considering differences in comments, whitespaces, and indentation.
- *Type 3*: Both tests contain a number of sequences that only differ in variable names and literal values, not considering differences in comments, whitespaces, and indentation.
- *Type 4*: Both tests are functionally the same for a different unit under test, independent of the syntax of either test. In other words, both tests verify the same property for a different class or function from the production code, independent of the syntax of either test.

Our dataset of classified clones, together with all scripts used in this work, are publicly available and shared on figshare: https://doi.org/10.6084/m9.figshare.c.4710692.v1.

F. Research Questions

Our comparison is driven by two research questions. In this section, we motivate why we investigate these research questions and explain the approach we use to answer them.

RQ1: *What are the characteristics of code clones in test code?*

Motivation: The quickest way for a developer to test a new feature is to copy, paste and modify an existing test [9]. We assume that because of this, in combination with the consistent structure of test code [10], a large amount of test code is duplicated. A recent case study on a large project from industry found that 49% of the entire test code are clones [11], which is significantly higher than the average of 7% to 23% for production code [12].

With this research question, we further investigate whether test code indeed contains a larger number of code clones than the average production code. Moreover, we investigate whether

the clones in test code inherently differ from those in production code.

Approach: To answer this research question, we calculate the clone density for each of the test suites in our dataset. Clone density (also known as clone percentage [21]) is defined as

$$clone\ density = \frac{f_c * 100}{f_{tot}}$$

where f_c denotes the number of cloned functions, and f_{tot} refers to the total number of functions in the test suite. In other words, the percentage of functions (i.e. tests) that appear in at least one clone fragment. Since we detect clones on a function level granularity, each clone fragment contains exactly one function. Therefore f_c is equal to the number of unique clone fragments. Once we have the clone density for each test suite, we can make a fair comparison with the averages found in production code.

We then inspect the clones found in the test suites to check whether they show traits that are specific to test code, and thus inherently differ from clones in production code. We illustrate these differences with a few representative examples.

RQ2: *How do clone detection tools perform on test code?*
Motivation: In order to assess and improve clone detection tools and techniques, clone benchmarks have been created that consist of open-source software projects together with a set of reference clones that appear in those projects. For example, Bellon's benchmark is the most used by the research community [29]. However, these benchmarks focus only on production code and do not contain test code [15]. As a result, clone detection tools and techniques are not being evaluated on test code, which might impact their effectiveness in detecting test code duplication.

With this research question, we investigate how code clone detection tools perform on test code. Moreover, we make our dataset publicly available, serving as a clone benchmark for test code.

Approach: To answer this research question, we calculate the precision and recall for each of the tools. Precision is defined as

$$precision = \frac{c_{TP} * 100}{c_{all}}$$

where c_{TP} denotes the number of true positive clones found by the tool and c_{all} the total number of clones found by the tool. Recall is defined as

$$recall = \frac{c_{all} * 100}{c_{tot}}$$

where c_{all} denotes the total number of clones found by the tool and c_{tot} the total number of clones in the dataset. However, since we do not know the total number of clones in the dataset, we approximate the recall by using all clones detected by the four tools as c_{tot}. This approximation is usually called the *relative recall*. Once we have these metrics, we can use them to compare the tools and evaluate their performance.

We then analyse the types of clones found by each tool. By calculating both the distibution of types and the relative recall per type for each tool, we can gain a better understanding of the impact that different clone detection techniques have on the types of clones that are detected.

V. Results and Discussion

In this section, we present our results and answer our research questions.

RQ1: *What are the characteristics of code clones in test code?*

Table II provides an overview of all clones detected by the four tools in each dataset. The left side of the table presents the number of clone pairs. We can already see that the number of clone pairs scales with the size of the dataset, with the larger Apache dataset containing 2,064 clone pairs, the medium Google dataset containing 406 clone pairs, and the small Patterns dataset containing only 74 clone pairs. In total, this results in 2,544 detected clone pairs.

The right side of Table II presents the number of individual clone fragments. We can use these, combined with the total number of functions from Table I, to calculate the clone density (i.e. the percentage of functions that are clone fragments). Since the number of clone pairs scales with the size of the dataset, the clone density is mostly the same for the different datasets, with the Apache dataset having a clone density of 29.1%, the Google dataset 23.2%, and the Patterns dataset 25.5%. As we can see, the clone density in our datasets of test code lies between 23% to 29%, which is higher than the average of 7% to 23% for production code.

An interesting observation can be made when comparing the number of clone pairs with the number of clone fragments. Because a clone pair consists of two clone fragments, we would assume that in general the number of clone fragments is greater than that of clone pairs. However, we can see that the Apache and Google datasets contain significantly fewer clone fragments than clone pairs, especially for type-2 and type-3 clones. This indicates that there are many large clone classes (e.g., sets of clone fragments that all form clone pairs between themselves). In other words, many tests are duplicated multiple times, each time only slightly modified.

The phenomenon of many large type-2 clone classes in test code is caused by the typical structure of unit tests. Figure 2 shows an example from the Apache dataset of such a typical type-2 clone in test code. As we can see, both tests are completely the same with exception of the input and the expected output of the unit under test. Since it is common practice to test multiple input values for each tested function, this kind of clone occurs a significant number of times in test code. For example, the specific clone fragments from Figure 2 are part of a clone class containing 10 such clone fragments, each exactly the same except for the input and expected output. These large clone classes inflate the number of clone pairs detected: consider the clone class a fully connected graph with 10 nodes (i.e. clone fragments), then each edge is a clone pair. And since the number of edges in a fully connected graph can be calculated as $\frac{n(n-1)}{2}$, this set of 10 clone fragments already causes 45 clone pairs. We can also see that these type-2 clones

TABLE II
OVERVIEW OF DATASETS.

Clone Type	Clone Pairs			Clone Fragments (Clone Density)		
	Apache	Google	Patterns	Apache	Google	Patterns
Type 1	48	24	4	55 (1.9%)	40 (3.2%)	5 (1.5%)
Type 2	708	310	36	373 (13.4%)	181 (14.7%)	40 (12.7%)
Type 3	954	32	13	214 (7.6%)	25 (2.0%)	15 (4.7%)
Type 4	354	40	21	170 (6.1%)	40 (3.2%)	20 (6.3%)
Total	2064	406	74	812 (29.1%)	286 (23.2%)	80 (25.5%)

```
public void testFormat() {
    BigFraction c = new BigFraction(1, 2);
    String expected = "1 / 2";
    String actual = properFormat.format(c);
    Assert.assertEquals(expected, actual);
    actual = improperFormat.format(c);
    Assert.assertEquals(expected, actual);
}
```

```
public void testFormatZero() {
    BigFraction c = new BigFraction(0, 1);
    String expected = "0 / 1";
    String actual = properFormat.format(c);
    Assert.assertEquals(expected, actual);
    actual = improperFormat.format(c);
    Assert.assertEquals(expected, actual);
}
```

Fig. 2. Example of a typical type-2 clone in test code, from the Apache dataset.

are the most common clone type in test code, contributing to around half of the clone density for each of the projects.

When observing the number of type-3 clone pairs detected, we notice that the Apache dataset contains 954 such pairs. Noticeably, the Google and Patterns dataset contain a significantly smaller proportion of type-3 clones. After inspection of these clones, we found that this oddity is caused by the nature of production code in the Apache Common's Math Library. This math library contains many implementations of mathematical algorithms, such as integration or interpolation, which are run on complex datastuctures, such as matrices or functions. The setup of these datastructures generally spans multiple lines. Thus, if the same algorithm is being tested with different inputs, these different inputs create a gap of several lines in the clone, resulting in a type-3 clone. This is also confirmed by the much lower number of type-3 clone fragments in the Apache dataset, which again indicate many large clone classes. We conclude that the large number of type-3 clones in the Apache dataset is also caused by the typical structure of unit tests.

Finally, we observe a non-negligible number of type-4 clones (or semantic clones). These semantic clones in test code inherently differ from those in production code since the semantics of a unit test differs from that of a function in production code. Specifically, we consider a test code clone pair as type-4 if both tests are functionally the same for a different unit under test, independent of the syntax of either test. In other words, both tests exercise a similar interface from different parts of a component library. These type-4 clones occur when the production code contains semantic clones which are tested with the same input values. Figure 3

```
public void testCopy_toStringBuilder_fromReadable() throws
    IOException {
  StringBuilder builder = new StringBuilder();
  long copied = CharStreams.copy(wrapAsGenericReadable(new
    StringReader(ASCII)), builder);
  assertEquals(ASCII, builder.toString());
  assertEquals(ASCII.length(), copied);
  StringBuilder builder2 = new StringBuilder();
  copied = CharStreams.copy(wrapAsGenericReadable(new
    StringReader(I18N)), builder2);
  assertEquals(I18N, builder2.toString());
  assertEquals(I18N.length(), copied);
}
```

```
public void testCopy_toWriter_fromReadable() throws
    IOException {
  StringWriter writer = new StringWriter();
  long copied = CharStreams.copy(wrapAsGenericReadable(new
    StringReader(ASCII)), writer);
  assertEquals(ASCII, writer.toString());
  assertEquals(ASCII.length(), copied);
  StringWriter writer2 = new StringWriter();
  copied = CharStreams.copy(wrapAsGenericReadable(new
    StringReader(I18N)), writer2);
  assertEquals(I18N, writer2.toString());
  assertEquals(I18N.length(), copied);
}
```

Fig. 3. Example of a type-4 clone in test code, from the Google dataset.

shows an example from the Google dataset of such a typical type-4 clone in test code. As we can see, the units under test `StringBuilder` and `StringWriter` are semantic clones from the production code as they contain the same functionality with a different implementation. When testing these units, both tests are semantically the same (i.e. verify the same properties) for different units. Note that, syntactically, these test can be considered type-3 clones, and are detected as such by most tools. However, when a clone pair satisfies the definition of multiple types, we classify as the type that applies to the most lines of code, which in this case is type 4.

> The typical structure of unit tests gives rise to many type-2 and type-3 clones. Many tests are duplicated multiple times, each time slightly modified to cover different configurations of input-output values. There is a non-negligible number of type-4 clones as well, caused by tests exercising a component library with similar interfaces.
>
> $\Longrightarrow$ Clones in test code are inherently different from clones in production code.

RQ2: *How do clone detection tools perform on test code?*

Table III provides an overview of the performance of the different clone detection tools. The left side of the table presents the precision of each tool per dataset. We can see that CPD

TABLE III
OVERVIEW OF CLONE DETECTOR PERFORMANCE.

Tool	Precision			Relative Recall		
	Apache	Google	Patterns	Apache	Google	Patterns
NiCad	100%	99%	100%	14.1%	27.8%	22.9%
CPD	100%	100%	100%	11.1%	30.5%	44.5%
iClones	98%	99%	100%	86.5%	47.5%	29.7%
TCORE	97%	95%	100%	30.8%	50.2%	36.4%

TABLE IV
CLONE PAIR CLASSIFICATION AND RELATIVE RECALL PER CLONE DETECTOR.

Clone Type	Clone Pairs (Distribution)				Relative Recall			
	NiCad	CPD	iClones	TCORE	NiCad	CPD	iClones	TCORE
Type 1	69 (16.3%)	55 (14.1%)	48 (2.3%)	59 (6.7%)	90.7%	72.3%	63.1%	77.6%
Type 2	162 (38.3%)	171 (44.0%)	651 (32.5%)	529 (60.9%)	15.3%	16.2%	61.7%	50.1%
Type 3	41 (9.7%)	4 (1.0%)	935 (46.7%)	120 (13.8%)	4.1%	0.4%	93.5%	12.0%
Type 4	149 (35.3%)	158 (40.7%)	348 (17.3%)	136 (15.6%)	35.9%	38.0%	83.8%	32.7%
Total	422 (100%)	388 (100%)	2001 (100%)	868 (100%)	16.5%	15.2%	78.6%	34.1%

has a precision of 100% for every datasets, meaning that it did not produce any false positives (e.g., incorrectly marking fragments of code as clones). NiCad has an almost perfect precision, with only a negligible number of false positives on the Google dataset. Both iClones and TCORE score slightly lower on precision, specifically for the larger datasets.

With the lowest precision being 95%, we conclude that clone detection tools perform excellent on test code in terms of precision.

It is interesting to note that the number of false positives is low, even though we configured the tools to detect smaller clones. More specifically, we used a minimum clone size of 5 LOC in contrast to the default 10 LOC. A lower minimum clone size naturally leads to an increased number of false positives, yet we notice that on our datasets of test code this is not the case. In fact, of all 1178 detected clone fragments, 474 (40.2%) are smaller than 10 LOC.

We deduce that test code contains smaller clones than production code. Therefore, general purpose tools should be configured accordingly when applied on test code.

The right side of Table III presents the relative recall of each tool per dataset. We can see that in general the relative recall is much lower than the precision. This is to be expected, since tool makers always have to consider the trade-off between precision and recall as an increase in recall generally also means an increase in false positives. The fact that all tools opt for a higher precision in exchange for a lower recall is favourable, as it is more important to detect a few qualitative clones rather than many false positives.

When observing the relative recall per dataset, we can see that the results differ between the datasets. The clones from the Apache dataset are mostly detected by iClones, with a relative recall of 86.5%, while other tools only detect up to 30% of the clones. Yet on the Google and Patterns datasets the relative recall of all tools varies between 22% and 50%. This indicates that the kind of code clones in the Apache dataset differs from the Google and Patterns datasets, and that some tools are better at detecting certain kinds of clones.

The percentage of false negatives (e.g., percentage of clones not found by a tool) can be easily calculated by taking the difference of the total (100%) and the relative recall. We can see that each tool suffers at least 50% in false negatives for the Google and Patterns datasets. The fact that the individual tools are only able to detect up to half of the clones detected by all tools shows that there is little overlap in the clones detected by the different tools. This further confirms that different tools are better at detecting certain kinds of clones.

The large number of false negatives combined with the earlier observation of large clone classes is worrisome. A test engineer wants to detect *all* copies of a certain code fragment when searching for test smells to assess which tests are copied most frequently and thus are the best refactoring candidates. Moreover, when tests are refactored, a test engineer wants to identify *all* tests that will be affected by the refactoring. In both cases, false negatives impair the refactoring process.

Clone detection tools currently suffer from many false negatives, which negatively impacts test refactoring. Further research towards clone detection on test code is warranted.

Table IV provides a detailed look at the clones detected by each tool. The left side of the table presents the number of clone pairs detected per type. It also shows the distribution of the detected clones over the types. This is also visualized in Figure V to provide an easier overview. We can see that the tree-based clone detection tools find the largest number of clone pairs, with iClones detecting 2001 clone pairs and TCORE detecting 868 clone pairs. The text- and token-based approaches find less clones, with NiCad detecting 422 clone pairs and CPD detecting 388 clones pairs.

The difference between the two approaches is mainly caused by type-2 and type-3 clones, which the tree-based tools are able to detect more easily. However, when looking at the distribution of detected code clones over the different types, we can see that the text- and token-based approaches primarily find type-2

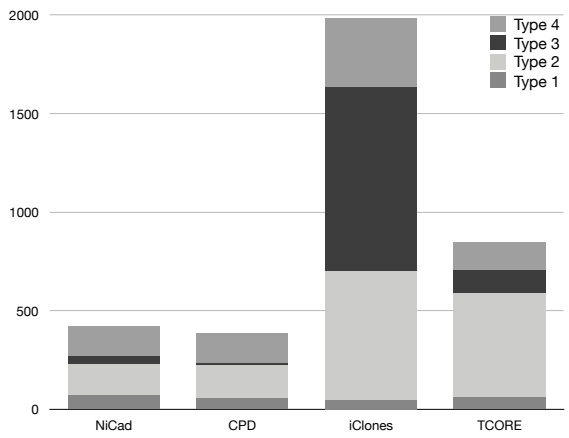

Fig. 4. Clone pairs detected by each tool per type.

clones, with around 40% of their detected clones being of type 2. Interestingly, they detect type-4 clones almost as often, with only slightly less than 40% of detected clones being of type 4. While the tree-based approaches generally detect more clones of every type, the distribution of these clones over the types is not the same. On the one hand, iClones detects primarily type-3 code clones, while the other types are proportionally all detected less than the text- and token-based approaches. On the other hand, TCORE detects primarily type-2 code clones, again with the other types all detected proportionally less than the text- and token-based approaches.

The right side of Table IV presents the relative recall for each type. While the relative recall of the different code clone detection tools on each dataset was already discussed previously, it is interesting to investigate the relative recall per type. NiCad's text-based approach performs the best for type-1 clones, with a relative recall of 90%. The tree-based approaches outperform the others for type-2 clones, with their relative recall over 50% compared to the 15% of the text- and token-based approaches. When it comes to detecting type-3 and type-4 clones, iClones clearly performs the best, with its relative recall being 93% and 83% respectively. All other approaches detect significantly less clones of these types, which causes iClones to have the highest overall relative recall (78.6%). The other tools on the other hand suffer from many false negatives: NiCad 83%, CPD 84%, and TCORE 65%. However, while it seems that iClones is the best option, it still suffers around 40% of false negatives for type-1 and type-2 clones.

Different clone detection tools detect different kinds of clones. Practitioners should combine multiple tools to achieve a more complete clone analysis. And when restricting to a single tool, make the choice based on the type of clones they are searching for, and not simply the tool that has the best overall precision or recall.

VI. Threats to Validity

A. Internal Validity

The manual classification of the discovered clones is a threat to internal validity. It is possible to make mistakes when performing manual classification on a large set of data. Moreover, there is room for interpretation when classifying code clones. To minimize this threat, we followed a strict set of rules during classification. Moreover, our dataset is publicly available to allow for review by the community.

A second threat to internal validity is the comparison of the different tools. We use relative recall as a metric during this comparison, since it is not feasible to calculate the actual recall. It is highly likely that there are more clones in the dataset than we detected, which would result in the actual recall being less than the reported relative recall. However, if there are additional clones in the dataset, none of the four tools used in our comparison detected them. Thus, recall of each tools would be lowered by the same amount. We can therefore safely use relative recall to compare clone detection tools. Moreover, the fact that actual recall is less than the relative recall strengthens our conclusions.

B. External Validity

In our evaluation, we ran four clone detection tools on three datasets. A threat to external validity is that the tools and the datasets we used in our evalution are not representative of all clone detection tools and test suites. To minimize this threat, we chose four tools that each implement a different clone detection algorithm and three datasets that vary in size and type. Future research can be performed to further confirm our findings, by adding more datasets and clone detection tools to our evaluation.

VII. Conclusion

In this paper, we report on an exploratory study concerning duplicated test code by running four clone detection tools (NiCad, CPD, iClones, and TCORE) on three open-source test suites (Apache, Google, and Patterns). In total, the four tools detect 2544 clone pairs in our dataset. This amounts to a clone density of 23% to 29%, which is higher than the average of 7% to 23% for production code.

We show that test code contains many large clone classes, especially of type 2 and type 3. In other words, many tests are duplicated multiple times, each time only slightly modified. Moreover, type-2 clones are the most common clone type in test code, contributing to around half of the clone density for each of the projects. This is caused by the typical structure of unit tests, which indicates that test clones are inherently different from production clones.

We demonstrate that code clone detection tools suffer from many false negatives when applied on test code. Even the tool that performs best on our dataset by achieving a relative recall of 78.6% still missed around 40% of type-1 and type-2 clones. Further research towards clone detection on test code is warranted in order to reliable apply them during test refactoring.

We also show that today practitioners can improve their clone detection process by (a) configuring their tools with a smaller minimum clone size, (b) using multiple tools to achieve a more complete clone analysis, and (c) choosing these tools depending on the type of clones they are interested in.

We made our dataset and all scripts publicly available. We hope that other researchers build upon this work by replicating this study with additional clone detection tools, and further extend the dataset. We also encourage clone detection tool makers to use it as a clone benchmark for test code.

VIII. Acknowledgments

This work is supported by (a) the ITEA TESTOMAT Project (number 16032), sponsored by VINNOVA – Sweden's innovation agency; (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

References

[1] K. Beck, *Test-driven Development: By Example*, ser. Kent Beck signature book. Addison-Wesley, 2003.

[2] G. Booch, *Object Oriented Design: With Applications*. Benjamin/Cummings Pub., 1991.

[3] M. Fowler and M. Foemmel, "Continuous integration," Thoughtworks, Tech. Rep., 2006.

[4] J. D. McGregor, "Test early, test often," *Journal of Object Technology*, vol. 6, no. 4, pp. 7–14, May 2007, (column). [Online]. Available: http://dx.doi.org/10.5381/jot.2007.6.4.c1

[5] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[6] L. Crispin and J. Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2009.

[7] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[8] V. Garousi and B. Kucuk, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52 – 81, 2018.

[9] H. Li, A. Lindberg, A. Schumacher, and S. Thompson, "Improving your test code with wrangler," School of Computing, University of Kent, Tech. Rep., 2009.

[10] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[11] W. Hasanain, Y. Labiche, and S. Eldh, "An analysis of complex industrial test code using clone analysis," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 482–489.

[12] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[13] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 56–65.

[14] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[15] C. K. Roy and J. R. Cordy, "Benchmarks for software clone detection: A ten-year retrospective," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 26–37.

[16] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, 2015.

[17] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International static analysis symposium*. Springer, 2001, pp. 40–56.

[18] J. Krinke, "Identifying similar code with program dependence graphs," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 301–309.

[19] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 301–310.

[20] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 81–92.

[21] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 368–377.

[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[23] 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*, vol. 28, no. 7, pp. 654–670, 2002.

[24] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th iEEE international conference on program comprehension*. IEEE, 2008, pp. 172–181.

[25] J. R. Cordy and C. K. Roy, "The NiCad clone detector," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 219–220.

[26] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 253–262.

[27] N. Göde and R. Koschke, "Incremental clone detection," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 219–228.

[28] B. van Bladel and S. Demeyer, "A novel approach for detecting type-iv clones in test code," in *2019 IEEE 13th International Workshop on Software Clones (IWSC)*. IEEE, 2019, pp. 8–12.

[29] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 577–591, 2007.