A comparative study of test code clones and production code clones

Reference:
van Bladel Brent, Demeyer Serge.- A comparative study of test code clones and production code clones
Journal of systems and software - ISSN 0164-1212 - 176(2021), 110940
Full text (Publisher's DOI): https://doi.org/10.1016/J.JSS.2021.110940
To cite this reference: https://hdl.handle.net/10067/1777240151162165141

A Comparative Study of Test Code Clones and Production Code Clones

Brent van Bladel[a], Serge Demeyer[a,b]

[a] *University of Antwerp, Belgium*
[b] *Flanders Make vzw, Belgium*

Abstract

Clones are one of the most widespread code smells, known to negatively affect the evolution of software systems. While there is a large body of research on detecting, managing, and refactoring clones in production code, clones in test code are often neglected in today's literature. In this paper we provide empirical evidence that further research on clones in test code is warranted. By analysing the clones in five representative open-source systems and comparing production code clones to test code clones, we observe that test code contains twice as many clones as production code. A detailed analysis reveals that most test clones are of Type II and Type III, and that many tests are duplicated multiple times with slight modifications. Moreover, current clone detection tools suffer from false negatives, and that this occurs more frequently in test code than in production code (NiCad = 76%, CPD = 90%, iClones = 12%). So even from a tools perspective, specific fine-tuning for test code is needed.

Keywords: software clones, unit-tests, clone detection

1. 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. The prevalence of clones in test code has been shown by Hasanain et al. [11] and was confirmed in our prior work [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 explicitly compare the clones in the test code against the clones in the system under test by running three state-of-the-art clone detection tools (NiCad, CPD, and iClones) on five representative open-source software projects. We extend our previous work [12] and classify, analyse, and compare a total of 21,289 test clones with 18,493 clones from production code. The dataset is made publicly available (see DOI 10.6084/m9.figshare.12644921) and classifies all clones in the appropriate categories (by type, tool, production or test code). The scripts to automatically create the dataset for a given system are publicly available as well.

Based on a quantitative and qualitative comparison of the clones in the test code against the clones in the system under test we make the following observations.

1. For each project, test code contains more than twice as much duplication as production code, even for projects with small amounts of clones.
2. Tests are often copied multiple times, each time with small modifications.
3. Clones in production code cause a significant increase in test clones.
4. Clones in test code are inherently different from clones in production code due to the typical structure of unit tests.
5. Current clone detection tools suffer from false negatives, and this occurs more frequently in test code than in production code, especially for text-based and token-based clone detectors.
6. A tree-based clone detection technique generally performs best on test code, while on production code it depends on the project.

We conclude that from a research perspective, more work on clones in test code is warranted. From a tools perspective, specific fine-tuning for test code is needed.
The remainder of this paper is organised as follows. Section 2 provides the required theoretical background while Section 3 lists the related work. Section 4 describes the experimental set-up, which naturally leads to Section 5 reporting the results. Section 6 lists avenues for further work, both from a research and tool builders perspective. Section 7 enumerates the threats to validity, and Section 8 concludes the paper.

2. Background

Code clone. When two fragments of 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, and 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.

Clone pair. When we consider a code clone that consists of exactly two clone fragments, we use the term clone pair. Most 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 (I, II, III, IV). 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 I clones are exactly the same, only allowing differences in comments, whitespaces, and indentation. Type II clones are the same as Type I clones, but also allow differences in variable names and literal values. Type III clones are the same as Type II clones, but also allow for lines of code to be added or removed in the clone fragment. Note that it is not required for these types of clones to be functionally similar. Semantic clones on the other hand are code clones that are semantically similar without necessarily being syntactically similar. They are often called Type IV clones.

3. Related Work

Clone Benchmarks. A lot of research has already been performed on software clones. In 2007, Koschke performed a survey of the literature on software clones [14]. This 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 [15]. Svajlenko et al. manually curated a data set containing six million inter-project clones (Type I, II, III, and IV), including various strengths of Type III similarity (strong, moderate, weak) [16]. Over the years, 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; test code is rarely ever considered separately [14, 15, 17].

Evaluation Criteria. A common denominator in comparative research on software clones is to evaluate across the different clone types (I, II, III), ignoring Type IV as most tools cannot identify them [14, 15, 17]. When analysing the prevalence of clones this is usually done by comparing the *clone density* (also known as clone percentage [18], or TCMp or TCLOCp depending on the granularity [19, 20]). When researchers compare clone detection tools, they calculate the precision and recall [15]. However, since it is impossible to know all clones in a given system, researchers typically approximate the recall by using the clones detected by all tools under study as a total. This is known as the *relative recall* [21].

Test Smells. 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 2018, Garousi et al. performed an extensive literature study on test smells, including knowledge from both industry and academia. Besides the work performed by Bavota et al., they found 37 sources that explicitly discuss negative consequences as a result of test smells [8]. Most recently, in 2020, Junior et al. conducted a survey amongst professionals to identify whether professional experience influences the adoption of test smells. They found that all developers introduce test smells irregardless of the developers experience [22].

Test Clones. 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 [23]. 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 our previous work, we performed 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 showed the prevalence of clones in test code and provided anecdotal evidence that these clones stem from the typical structure of unit tests [12].

There is a large body of work investigating the prevalence and characteristics of software clones across the different clone types (I, II, III). *Clone density* is a commonly applied metric when comparing clones within software systems, while both *precision* and *relative recall* is used when comparing clone tools. It is only recently that clones in test code are investigated as a separate topic. At the time of writing, there is no research investigating the differences between clones in test code and clones in production code.

4. 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.

4.1. Clone Detection Tools

There are many different code clone detection tools available, 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 [15].

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 [24, 25]. Other techniques include static analysis of memory states at each procedure exit point [26], or applying random testing to detect similar function output [27]. More recently, machine learning techniques, such as deep neutral networks, have been successfully used to detect more difficult Type III and Type IV clones [28, 29].

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* [18] 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 4.3). For example, *Deckard* [30] 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* [31], SourcererCC [32], and CloneWorks [33] were considered, but their output was not easily converted to our reference format (see Section 4.4).
- *Approach and implementation*: To allow for a more broad analysis, we selected tools with different approaches: one text-based, one token-based, and one token/tree-based hybrid. We also selected tools with different implementations: one academic tool, one open-source tool, and one commercial tool. We would have also liked to include a PDG-based approach, but we were unable to find one that works for both production and test code. For example, *TCORE* [34] was considered, but it can only be used on test code. We did not select tools that implement the

more advanced techniques, such as memory states or machine learning, as they typically focus on Type IV clones.

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

- *NiCad* is an academic tool that 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 form. Finally, the normalized fragments are linewise compared using an optimized longest common subsequence algorithm to detect clones [35, 36].
- *PMD's CPD* is an open-source tool that adopts a token-based approach based on the Karp–Rabin string matching algorithm on a frequency table of tokens in order to detect clones [15].
- *iClones* is a commercial tool that 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 [37, 38].

4.2. Dataset

For our comparison, we selected five open-source Java projects from GitHub: the Java Spring Framework (from now on referred to as Spring), the Elastic Search distributed search engine (Search), the Apache Commons Math library (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 five projects make use of a continuous integration (CI) server that runs the test suite after each commit. At the time of analysis[1], all projects pass their CI build.

We use both the production code and the test suite of these projects as the dataset for our comparison. Table 1 shows an overview of the size of each project in terms of functions (for the production code), tests (for the test code), and lines of code (LOC). Note that the LOC metric does not include comments or blank lines. The Spring dataset is the largest of the five with a total of 627k LOC, the Patterns dataset is the smallest with 28k LOC. We selected the projects specifically to have this difference in size to allow for more generalized results.

[1] May 2020

Table 1: Dataset descriptive statistics.

Name		Functions	LOC	Min	Median	Max
Spring	Production	18,821	295,232	6	11	429
	Test	12,105	331,852	6	9	165
Search	Production	28,911	236,239	8	13	650
	Test	9,401	145,041	8	17	389
Apache	Production	7,564	92,683	1	5	848
	Test	6,791	95,562	1	8	359
Google	Production	6,303	87,716	1	3	122
	Test	8,917	112,821	1	7	604
Patterns	Production	1,690	17,962	2	3	66
	Test	603	10,990	1	7	43

To allow for comparison between production clones and test clones, we consider the production code and test code of each project as separate datasets. This means that all detected clones are completely contained within either the production code or test code of one project.

4.3. 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 allows us to match the same clone detected by multiple tools, since the start and end of the clone is 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 when considering test code.

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 test level granularity in the test code, we choose to decrease the minimum clone length. For fair comparison, we do this for both production code and test code. By default, the 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

```
<clone type=”T3” iclones=”True” pmd=”False” nicad=”False”>
  <source file=”SpringFramework/production/CollectionToArrayConverter.java”
    startline=”65” endline=”80”>
  </source>
  <source file=”SpringFramework/production/StringToArrayConverter.java”
    startline=”61” endline=”76”>
  </source>
</clone>
```

Figure 1: Example of a clone pair in the reference XML format.

code, as this allows for the smaller duplicated tests to be detected without generating many false positives [34]. Therefore, we set the minimum clone size parameter for NiCad, iClones, and CPD to half their default.

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.

4.4. Postprocessing

After running each clone detection tool on the dataset, we have a set of XML files containing the detected clones. To allow for easy analysis, we use a Python script to merge the XML output of the different tools into a single file. Figure 1 shows the format of this merged XML file. As shown in the figure, we represent clone pairs using the location of each fragment (i.e. the filename, startline, and endline of the clone fragments). We also add three boolean attributes for each clone; one for each tool indicating whether or not the tool was able to detect the clone. Finally, the type attribute is added after classification (see Section 4.5).

4.5. Classification

After postprocessing, we performed type classification on all detected clones. Due to the large amount of detected clones, we partially automated this classification using a Python script. For each clone pair, this script: (1) extracts both code fragments from the source code, (2) normalizes indentation and removes comments from the code fragments, and (3) compares the fragments. If the comparison shows that a continuous sequence of at least 5 lines from one fragment is exactly the same as a continuous sequence in the other fragment, the clone is automatically classified as a Type I clone. Otherwise, if there is a matching continuous sequence of at least 5 lines, but the sequence differs in variable names and/or literal values, the clone is automatically classified as a Type II clone. Finally, if one of the fragments contains a matching continuous sequence of at least 5 lines only differing in variable names and/or literal values, but the matched lines in the other fragment is not continues, the clone is automatically classified as a Type III clone.

When the script cannot classify the clone according to any of these rules, it shows both normalized fragments side by side in a GUI and asks the user to manually classify the clone. In practice, this means deciding wether the clone is of Type III or a false positive. Due to this limited human interaction, a consistent classification is guaranteed and any room for interpretation is removed.

Type IV clones are not considered, as the detection tools are focussed on syntactic similarity. Moreover, since the semantics of test code differ from the semantics of production code, it would be difficult to make a meaningful comparison between the two.

4.6. Research Questions

In this paper we explicitly compare the clones in production code against the clones in test code across the different clone types (I, II, III). We do so from a system-oriented and a tool-oriented perspective. From the system-oriented perspective we investigate characteristics of the clones within the same system and analyse the nature of the differences. From the tool-oriented perspective, we compare the precision and recall and see whether there are differences when applied on production code versus test code. As such our comparison is driven by four research questions. In this section, we motivate why we investigate these research questions and explain the approach we use to answer them.

RQ1: *What is the difference in clone density for production code and test code?*
Motivation: A recent case study on a large project from industry found that 49% of the entire test code is duplicated [11]. Our previous work confirmed the prevalence of clones in test code by analyzing three open-source systems [12]. However, it is yet unknown whether test code contains more or fewer clones than its production counterpart.

Approach: To answer this research question, we calculate the clone density for each of the datasets. Clone density (also known as clone percentage [18], or TCMp or TCLOCp depending on the granularity [19, 20]) 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 system. In other words, the percentage of functions (or 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 dataset, we can make a comparison between production code and test code.

We inspect the distribution of clone types in each dataset by calculating the clone density for each type. In other words, for each clone type we calculate the percentage of duplication in the entire project as if the clones of other types did not exist. We expect that, when taking two subsets of a software system, the distribution of the different clone types should be relatively constant in both. By making the comparison between production code and test code, which are two subsets of one software system, we can verify whether this still holds or whether test clones show traits that are specific to test code, and thus inherently differ from clones in production code.

RQ2: *How do clone classes in test code differ from clone classes in production code?*
Motivation: 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. By investigating clone classes, we can determine how often the same unit tests is cloned and analyse the extent of the differences between each clone fragment.
Approach: We compare the amount of *clone classes* in production code and test code, where a clone class consists of all clone fragments that form clone pairs between themselves. We explicitly distinguish between the different clone types (I, II, III).

To verify whether the relation between clone classes and test code is significant, we calculate the Jaccard Similarity Coefficient. We use Jaccard similarity since it is best suited for binary data (e.g. Production code or Test code; clone pair or clone class) [39, 40]. The Jaccard Similarity Coefficient (JSM) is defined as

$$JSM = \frac{|X \cap Y|}{|X \cup Y|}$$

where, in our case, X denotes the set of clone fragments from test code and Y denotes the set of clone fragments that are part of a clone class. Note that if a clone fragment is an element of Y and not an element of X, it is a clone fragment from production code that forms a clone class. Similarly, if a clone fragment is an element of X and not an element of Y, it is a clone fragment from test code that forms a clone pair but no clone class. As a result, a higher JSM indicates that code classes occur more often in test code than in production code.

RQ3: *How can the differences between test clones and production clones be explained?*
Motivation: Kapser et al. argued that not all clones are harmful, and proposed several patterns of accepted cloning behaviour [41]. One such pattern is *templating*, a matter of parameterisation of a proven solution. *API/Library Protocols* are a particular instance of templating, inducing a sequence of procedure calls to achieve the desired behaviour. The consistent structure of unit test code (the *setup-stimulate-verify-teardown* (S-S-V-T) cycle [10]) may explain why clones occur so often in test code.
Approach: We collect anecdotal evidence for typical examples of both Type II and Type III test clones from our dataset. We investigate whether the consistent structure of test code causes these clones, and we look into the relationship between these clones and the production code under test.

RQ4: *How effective are clone detection tools on test code compared to production code?*
Motivation: In order to assess and improve clone detection tools and techniques, clone benchmarks have been created [42, 16]. However, these benchmarks focus only on production code and do not contain test code [17]. 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.

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 three tools as c_{tot}. This approximation is called the *relative recall* and is commonly used in the field of information retrieval as an upperbound approximation of the actual recall [21]. The relative recall also provides us with the percentage of false negatives, since it can be calculated as

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

Once we have these metrics, we can use them to compare the tools and evaluate their performance.

We then analyse the types of clones (Type I, Type II, Type III) found by each tool. By calculating both the distribution 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 (not) detected. Moreover, we investigate how the characteristics of test clones affect the results of code clone detection tools.

5. Results and Discussion

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

RQ1: *What is the difference in clone density for production code and test code?*

Table 2 provides an overview of all clones detected by the three tools for each dataset in terms of clone pairs, clone fragments, and clone density. In

Table 2: Overview of clone pairs, clone fragments, and clone density.
(For the clone density columns, the minimum is underlined, the maximum is double underlined.)

	Production code			**Test code**		
Project	**Pairs**	**Frag.**	**Density**	**Pairs**	**Frag.**	**Density**
Spring	4,441	867	4.6%	2,862	1,759	14.5%
Search	11,422	1,445	5.0%	9,881	2,238	23.8%
Apache	1,554	1,194	15.8%	5,327	2,103	31.0%
Google	851	446	7.1%	3,163	1,934	21.7%
Patterns	225	67	4.9%	56	77	12.8%
Total	18,493	4,019	-	21,289	8,111	-

total 39,782 clones were detected, of which 18,493 in production code and 21,289 in test code.

When considering the clone density (e.g. the duplication relative to the size of the code), we can see that the production code of all projects contain around 5% duplication, where Apache is a notable exception with 15.8%. These results fall within the average duplication of 5% - 20% reported in literature [43]. If we consider the clone density of the test code, we can see that the Search, Apache, and Google datasets exceed this average of 5% - 20% duplication, with 23.8%, 31%, and 21.7% respectively. The Spring and Patterns projects have less duplication in their test code, with a clone density of 14.5% and 12.8% respectively. However, these were also the projects with the lowest amount of clones in the production code, with the clone density in test code still more than twice of that in production code.

Test code contains twice as many clones as production code, regardless of the system analysed.
⇒ Developers clone twice as often in test code than in production code.

Table 3 provides an overview of the clone density per type (Type I, Type II, Type III). More specifically, it shows for each project the clone density when only considering duplication of a certain type. For exact clones (Type I), the clone density in both production code and test code is minimal (i.e. between 0% - 2%). For Type II and Type III clones, the clone density is between 1% - 7% in production code and between 5% - 15% in test code. Whether there are more Type II clones or Type III clones depends on the project. However, for each project, the clone density of these non-exact clones is consistently higher in the test code compared to the production code. This causes the overall increase in test code clone density compared to production code.

Table 3: Overview of clone density per type.

(The minimum in each column is underlined, the maximum is double underlined.)

	Production code			Test code		
Project	**Type I**	**Type II**	**Type III**	**Type I**	**Type II**	**Type III**
Spring	0.4%	2.3%	1.8%	1.3%	6.8%	6.5%
Search	0.7%	1.8%	2.5%	0.8%	8.0%	15.0%
Apache	2.1%	6.7%	7.0%	1.9%	15.5%	13.6%
Google	0.7%	4.8%	1.5%	1.6%	12.8%	7.3%
Patterns	0.4%	2.2%	1.4%	0.0%	8.5%	4.3%

Table 4: Overview of clone classes.

	Production code		Test code	
Project	**Classes**	**Fragments**	**Classes**	**Fragments**
Spring	73	49.1%	234	69.0%
Search	109	66.8%	229	76.1%
Apache	135	56.4%	240	70.7%
Google	39	59.0%	214	59.2%
Patterns	11	83.6%	11	51.9%

Test code has increased amounts of Type II and Type III clones compared to production code.
⇒ When developers clone test code, they make small modifications.

RQ2: *How do clone classes in test code differ from clone classes in production code?*

Table 4 provides an overview of the clone classes detected by the three tools in each dataset. The column labelled *Classes* shows the amount of clone classes that contain at least 3 clone fragments. Therefore clone pairs (e.g. clone classes with only 2 fragments) are not counted. The column labelled *Fragments* shows the percentage of clone fragments that are part of these clone classes. In other words, it is the percentage of code fragments that is duplicated more than once throughout the project.

For production code, the percentage of code fragments that is duplicated more than once is around 50% - 65%. For test code, this is around 60%-75%. We see that the amount of such fragments is consistently higher for test code compared to production code, most significantly in the Spring, Search, and Apache datasets. The Patterns dataset is a notable exception, however since it only has 11 clone classes (the smallest in our analysis) for both its production and test code, the sample size is too small to draw meaningful conclusions.

To verify whether the difference in the amount of clone classes between test code and production code is significant, we calculate the Jaccard Similarity Coefficient (JSM). It's a measure of similarity for two sets of data (clone fragments that are part of a clone class and clone fragments that are part of the test code): the higher the percentage, the more similar the two sets are. For context, if we were to generate datapoints according to a uniform random distribution, the JSM would equal 33.33%. On the other hand, if clone classes occurred only in test code and clone pairs only in production code, the JSM would be equal to 100%. In our case, we arrived at a JSM of 52.8%, indicating a significantly higher amount of clone classes in test code compared to production code.

Figure 2 shows the boxplot of the size of the clone classes for each datasets. Or, in other words, it shows the amount of times a clone fragment was duplicated when it was duplicated more than once. As we can see, the average size of the clone classes is 4 across all datasets, both for test code and production code. The exception here is the Google dataset, having an average of 6 clone fragments per class. Outliers are not visualized on this graph as there are too many to provide an informative graphic. Instead, to elaborate on the extremes, Figure 3 shows an overview of the size of clone classes across the whole dataset. Again, the size of the clone classes for both production code and test code is similar in general. However, the production dataset count 34 outliers while the test datasets on the other hand count 87 outliers. These outliers are instances where a clone fragment was duplicated more than 10 ten times, which occurs more than twice as often in test code. The most extreme case was from the Search dataset, where the same test was duplicated and slightly modified 178 times. This indicates that some clone fragments in test code are copied far more often than production code.

Clone classes in test code are larger compared to production code, as clone fragments in test code are duplicated more often.
⇒ Tests are often copied multiple times.

RQ3: *How can the differences between test clones and production clones be explained?*

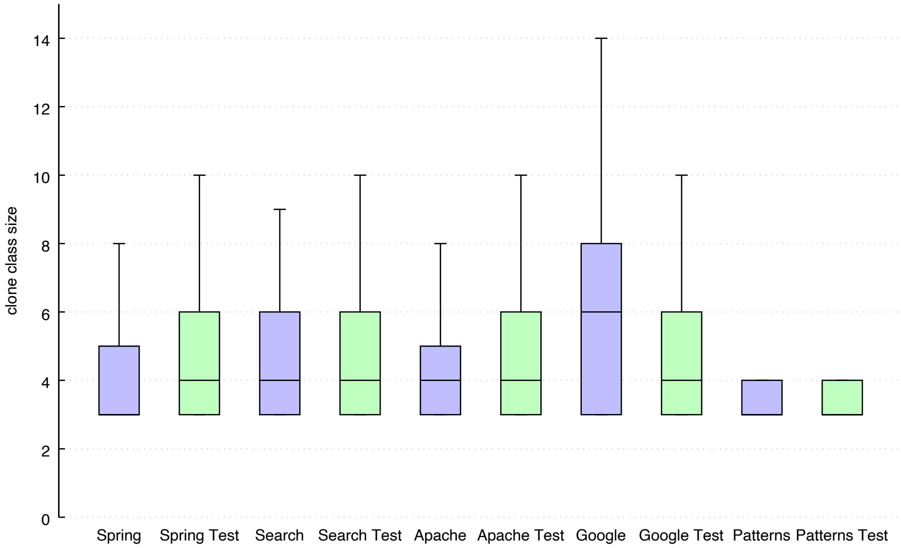

Figure 2: Clone class sizes for each dataset.

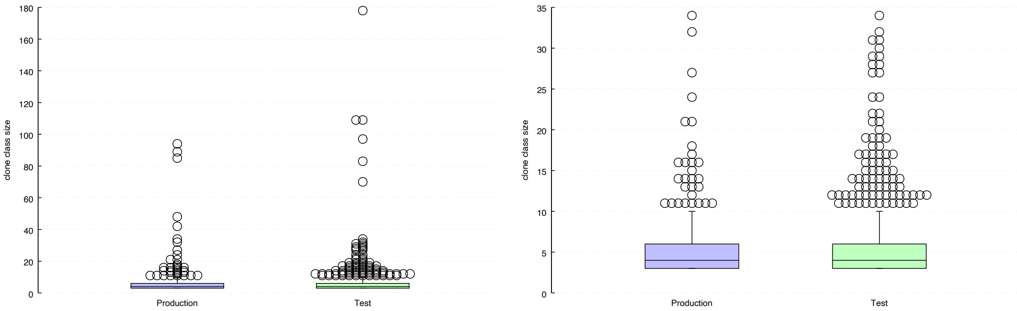

Figure 3: Clone class sizes for production code and test code, with oultiers.

(The left figure shows the full overview, while the right figure zooms in on the central clusters.)

We have shown that test code contains more Type II and Type III clones than production code, and that clone fragments in test code are often duplicated multiple times. The phenomenon of many larger Type II clone classes in test code is caused by the typical structure of unit tests. Figure 4 shows an example from the Search test dataset of such a typical Type II 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 function, this kind of clone occurs a significant number of times in test code. For example, the specific clone fragments from Figure 4 are part of a clone class containing 32 such clone fragments, each exactly the same except for the input and expected output.

```
public void testFoldingToLocalExecWithProject() {
    PhysicalPlan p = plan(”SELECT keyword FROM test WHERE 1 = 2”);
    assertEquals(LocalExec.class, p.getClass());
    LocalExec le = (LocalExec) p;
    assertEquals(EmptyExecutable.class, le.executable().getClass());
    EmptyExecutable ee = (EmptyExecutable) le.executable();
    assertEquals(1, ee.output().size());
    assertThat(ee.output().get(0).toString(), startsWith(”test.keyword{f}#”)
    );
}
```

```
public void testLocalExecWithPrunedFilterWithFunction() {
   PhysicalPlan p = plan(”SELECT E() FROM test WHERE PI() = 5”);
   assertEquals(LocalExec.class, p.getClass());
   LocalExec le = (LocalExec) p;
   assertEquals(EmptyExecutable.class, le.executable().getClass());
   EmptyExecutable ee = (EmptyExecutable) le.executable();
   assertEquals(1, ee.output().size());
   assertThat(ee.output().get(0).toString(), startsWith(”E(){r}#”));

}
```

Figure 4: Example of a typical Type II clone in test code, from the Search test dataset. (differences marked in red)

For the same reason, the typical structure of unit tests also cause many Type III clones. Figure 5 shows an example from the Spring test dataset of such a typical Type III clone in test code, which was copied 10 times with slight modifications. Here as well, the differences accommodate different input and expected output. However, in contrast to a Type II clone, in order to test different states of a class, additional calls on the object under test are inserted.

```
public void printScopedAttributeResult() throws Exception {
  tag.setExpression(”bean.method()”);

  int action = tag.doStartTag();
  assertThat(action).isEqualTo(Tag.EVAL_BODY_INCLUDE);
  action = tag.doEndTag();
  assertThat(action).isEqualTo(Tag.EVAL_PAGE);
  assertThat(((MockHttpServletResponse) context.getResponse()).
    getContentAsString()).isEqualTo(”foo”);
}
```

```
public void printHtmlEscapedAttributeResult() throws Exception {
  tag.setExpression(”bean.html()”);
  tag.setHtmlEscape(true);

  int action = tag.doStartTag();
  assertThat(action).isEqualTo(Tag.EVAL_BODY_INCLUDE);
  action = tag.doEndTag();
  assertThat(action).isEqualTo(Tag.EVAL_PAGE);
  assertThat(((MockHttpServletResponse) context.getResponse()).
    getContentAsString()).isEqualTo(”&lt;p&gt;”);
}
```

Figure 5: Example of a typical Type III clone in test code, from the Spring test dataset.
(differences marked in red)

Test code is often duplicated multiple times with slight changes in order to test different inputs of a function (Type II) or different states of a class (Type III). These test clones can be seen as a particular instance of templating, and are therefore not necessarily harmful.
⇒ The typical structure of unit tests gives rise to many Type II and Type III clones.

As we have seen, a function in production code can lead to Type II clones in test code. While this can be seen as a instance of unharmful templating, it can have further consequences. Namely, if a function in production code is duplicated, we found that its tests are duplicated as well. Figure 6 provides a generalized example. Here, function A has three tests that check different inputs for the function. As a result, they are Type II clones. Function B, which is a clone of function A, is being tested with in the same way. Not only are the tests of function B Type II clones, every test of function A is now a Type III clone with every test of function B, as they only differ in the function being called (and possibly the input value).

We found many cases of this scenario in our dataset. Most notably in the Apache dataset, which implements different algorithms for mathematical

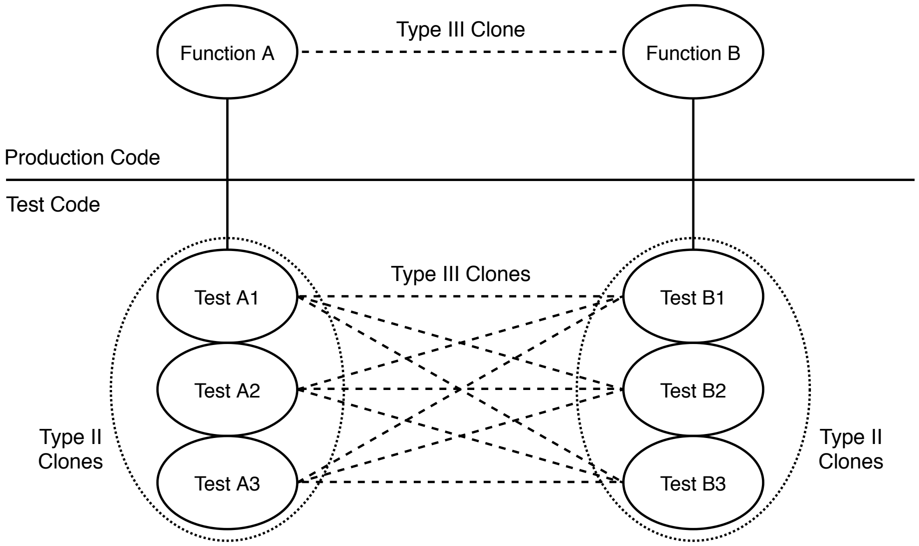

Figure 6: Relation between test clones and production clones.

problems such as integration and interpolation. The tests of each of these algorithms only differ in the call to the algorithm, causing them to all be Type III clones. Another example is from the Patterns dataset, which contains multiple *dummy* classes to showcase their design patterns. Each of these classes provides a similar interface, which again causes duplication in the test code.

When functionality is duplicated in production code, all tests that verify this functionality are also duplicated.
⇒ Clones in production code cause a significant increase in test clones.

RQ4: *How effective are clone detection tools on test code compared to production code?*

Table 5 provides an overview of the precision and the relative recall of the different clone detectors on each dataset. When we look at the precision, all tools produce few false positives (e.g. incorrectly marking fragments of code as clones), with a total precision between 95% - 100%. All three clone detection techniques (text-based, token-based, and tree-based) are capable of detecting clones with a high precision and this both for production and test code.

Table 5: Overview of the precision and relative recall for each clone detector.

		Precision			Relative Recall		
Project		**NiCad**	**CPD**	**iClones**	**NiCad**	**CPD**	**iClones**
Spring	**Production**	95%	97%	99%	93%	8%	9%
	Test	100%	100%	100%	28%	13%	88%
Search	**Production**	100%	99%	100	70%	21%	31%%
	Test	100%	99%	98%	9%	4%	98%
Apache	**Production**	88%	99%	98%	46%	17%	83%
	Test	91%	98%	99%	30%	11%	93%
Google	**Production**	87%	100%	100%	84%	31%	40%
	Test	98%	99%	98%	55%	23%	49%
Patterns	**Production**	88%	97%	100%	27%	16%	91%
	Test	84%	90%	90%	29%	46%	50%
Total	**Production**	97%	99%	100%	74%	18%	31%
	Test	96%	99%	98%	24%	10%	88%

When we look at relative recall, we can see that NiCad's text-based technique detects the most clones in production code, with a total relative recall of 74%. Although iClones does perform well on the Apache and Patterns production datasets, the total relative recall of both PMD's token-based approach and iClones' tree-based approach is significantly less, with 18% and 31% respectively. When considering the test code datasets, however, we note that iClones tree-based approach outperforms the others with a total relative recall of 88%. Moreover, with exception of the Patterns dataset, the tree-based approach performs consistently better on test code compared to production code across all projects.

The reason why a tree-based approach works better is caused by the prevalence of Type III clones in test code. This can be deduced from Figure 7, which provides an overview of the total amount of clones detected, classified per type (Type I, Type II and Type III) and per tool. NiCad and CPD have a very similar distribution, with around 12% of their detected clones being of Type I, 66% of Type II, and 21% of Type III. In relative terms, iClones detects fewer Type I clones (4%) and Type-II clones (26%), but a lot more Type III clones (70%). This confirms that the tree-based approach can detect Type III clones more easily than a text- or token-based approach. The text- and token-based approaches are much better at detecting Type I and Type II clones on the other hand.

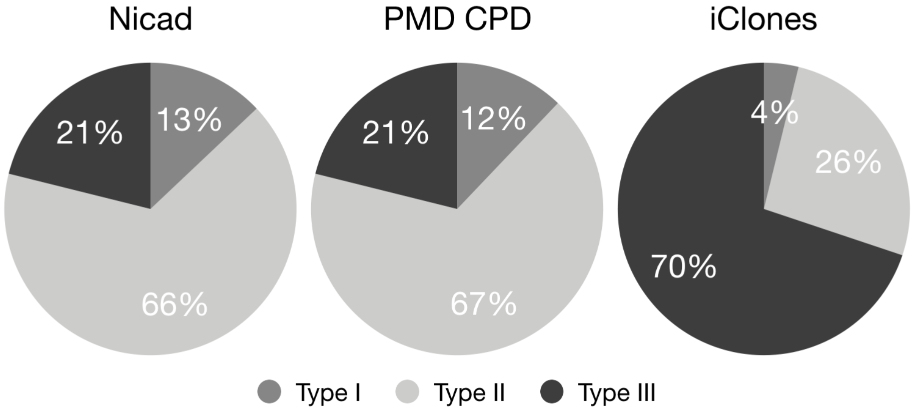

Figure 7: Clones of each type detected by the different tools.

Nevertheless, low relative recall indicates that every tool suffers from many false negatives. These false negatives occur more frequently in test code than in production code, especially for text-based and token-based clone detectors. For NiCad, the relative recall decreases from 74% on production code to 24% on test code (-50%). In other words, the amount of false negatives increases from 26% to 76%. Similarly, CPD sees a difference from 18% to 10% (-10%), and therefore an increase in false negatives from 82% to 90%. iClones however sees a positive difference: relative recall increases from 31% to 88% (+57%).

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.

We conclude that, even though the tools perform excellent in terms of precision, every tool suffers from false negatives (e.g. clones which are not detected). These false negatives occur more frequently in test code than in production code, especially for text-based and token-based clone detectors. When detecting clones in test code, a tree-based approach works better due to the larger amount of Type III clones.
⇒ From a tools perspective, specific fine-tuning for test code is needed.

6. Avenues for Further Research

In the paper we provide empirical evidence on the differences between clones in production code and test code. We argue that clones in test code are sufficiently different from clones in production code to warrant increased research attention. Below we sketch a few avenues for further research, refining existing work from the cloning community but gear it towards clones in test code.

Clone Genealogies. In 2005, Kim et al. coined the term "Code Clone Genealogies" for describing how a family of clone classes evolves over time [44]. They illustrated that clones are either short-lived and disappear due to natural code evolution, or long-lived and get changed consistently over time since there is no proper way to refactor them into a single abstraction. Krinke as well studied the changes applied to a clone class over time and noticed that clones are not always changed consistently [45]. Knowing that clone classes are often large (i.e. a single unit test gets copied multiple time) and consist mainly of Type II and Type III clones (i.e they consist of smaller variations), studying how such a clone class evolves over time would be interesting. Do these test clones appear in a short bursts or do they slowly emerge over time? In the former case, they are a potential symptom of a well thought out test case design covering a series of well-defined input-output combinations. In the latter case, they may illustrate graceful co-evolution between the system under test and its test cases. Another interesting avenue is to consider how test clones deal with stable or unstable APIs in the system under test [46].

Test Amplification. Test amplification is the act of automatically transforming a manually written unit-test to exercise boundary conditions [47]. In that sense, test amplification is a special kind of test generation: it relies on test cases previously written by developers which it tries to improve. DSpot is an example of a test amplification tool for Java projects [48] which has been replicated for Pharo/Smalltalk within our lab under the name of SmallAmp [49]. Such tools iteratively create extra test cases by changing the setup and the assertions, resulting in a new and larger set of test cases, essentially creating a series of Type III clones of the amplified test case. Yet, test amplification tools amplify a single test at a time and don't exploit the fact that some tests are copied multiple times. One could for instance focus the test amplification process on tests which are cloned often as they represent important hot spots in the system under test. Conversely, one could amplify tests which are never copied as these may represent less tested parts of the system.

Test Transplantation. Initially clones were mainly investigated from a single system perspective [42]. Yet, with the arrival of various open-source code hosting platforms, researchers investigated inter-project Type III clones as a way to search for idioms, patterns and API-usages [16, 50]. In a similar vein, if we would be able to find inter-project clones within test code, we could mine the "wisdom of the crowds" for testing a certain library or API. We could then go one step further and improve the test base from one system by applying a variant of code transplantation [51]. Rather than transplanting tests for clones in the system under test (like advocated by Zhang et al. [52]), we argue to transplant the cloned tests themselves.

Reduce False Negatives. In our prior work, we noticed that clones in test code tend to be smaller in size [12] Hence one cannot just blindly follow the default parameters provided in clone detection tools when searching for clones in test code. So an obvious way to reduce the amount of false negatives is fine-tuning the available parameters to better accomodate the nature of test code. Tools like SonarQube (`https://www.sonarqube.org`), CodeScene (`https://codescene.io`), and source{d} (`https://github.com/src-d`) in particular should explore this avenue. A next step would be to exploit the presence of the abstract syntax tee in tree-based clone detectors (which have the least amount of false positives anyway) to exploit the consistent structure of unit test code (the *setup-stimulate-verify-teardown* (S-S-V-T) cycle [10]).

Test Clone Management. One school of thought in the cloning community argues that non-harmful clones should be tolerated and that tool support should focus on managing the consistency between clones [53]. Several such consistent change recommenders have been created over the years, we list just a few: CloneTracker [54], CodeCloningEvents [55], Clone Change Notification [56], Clone Notifier [57]. Knowing that clones in test code often get copied multiple times, it would be interesting to explore how such recommender systems would work for test code.

7. Threats to Validity

7.1. Internal Validity

The classification of the discovered clones is a threat to internal validity. There is room for interpretation when manually classifying code clones.

To minimize this threat, we automated a large part of the classification, limiting manual classification to the decision between Type III and false positives. 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 tools used in our comparison detected them. Thus, recall of each tool would be lowered, which would not affect our conlusions.

7.2. External Validity

In our evaluation, we ran three clone detection tools on five open-source Java projects. 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/or code bases. To minimize this threat, we chose the tools such that they differ in implementation (open-source, academic, and commercial) and clone detection algorithm (text-, token-, and tree-based). Similarly, we chose the datasets such that they vary in size, type, and complexity. We encourage future research to confirm our findings by adding more datasets and clone detection tools to our evaluation. More specifically, we believe that extending the dataset with different programming languages (such as dynamically typed programming languages) and different tests (such as integration tests) are important to measure the generalizability of our results.

8. Conclusion

In the paper we provide empirical evidence on the differences between clones in production code and test code. We collected clone reports from five representative open-source software projects using three state-of-the-art clone detection tools (NiCad, CPD, and iClones). We then classified, analysed, and compared a total of 21,289 test clones with 18,493 clones from production code. We found that test code contains twice as many clones than production code, even for projects with a small amount of clones. This increase can be attributed to significantly more occurrences of Type II and Type III clones; Type I (exact) clones are negligible. We deduced that when developers clone test code, they they often copy multiple times making small modifications to test different input values.

The clone detection tools under analysis perform excellent in terms of precision, yet every tool suffers from false negatives (e.g. clones which are not detected). These false negatives occur more frequently in test code than in production code, especially for text-based and token-based clone detectors.

We conclude that from a research perspective, more work on clones in test code is warranted. Clone genealogies, test amplification, and test transplantation in particular seem promising avenues for future research. From a tools perspective, specific fine-tuning for test code is needed to reduce the number of false positives. Improving the current generation of tree-based clone detectors to exploit the consistent structure of unit test code is the way to go. Since clones in test code are often duplicated multiple times, clone management tools recommending consistent modifications to clones in test code are more than welcome.

9. 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, Kent Beck signature book, Addison-Wesley, 2003.

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

[3] M. Fowler, M. Foemmel, Continuous integration, Tech. rep., Thoughtworks (2006).

[4] J. D. McGregor, Test early, test often, Journal of Object Technology 6 (4) (2007) 7–14, (column). doi:10.5381/jot.2007.6.4.c1.
URL http://dx.doi.org/10.5381/jot.2007.6.4.c1

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

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

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

[8] V. Garousi, B. Kucuk, Smells in software test code: A survey of knowledge in industry and academia, Journal of Systems and Software 138 (2018) 52 – 81. doi:https://doi.org/10.1016/j.jss.2017.12.013.

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

[10] B. Van Rompaey, B. Du Bois, S. Demeyer, M. Rieger, On the detection of test smells: A metrics-based approach for general fixture and eager test, IEEE Transactions on Software Engineering 33 (12) (2007) 800–817.

[11] W. Hasanain, Y. Labiche, 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] B. van Bladel, S. Demeyer, Clone detection in test code: An empirical evaluation, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2020, pp. 492–500.

[13] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, 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] R. Koschke, Survey of research on software clones, in: Dagstuhl Seminar Proceedings, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[15] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of computer programming 74 (7) (2009) 470–495.

[16] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, M. M. Mia, Towards a big data curated benchmark of inter-project code clones, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 476–480.

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

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

[19] C. K. Roy, J. R. Cordy, An empirical study of function clones in open source software, in: 2008 15th Working Conference on Reverse Engineering, IEEE, 2008, pp. 81–90.

[20] C. K. Roy, J. R. Cordy, Near-miss function clones in open source software: an empirical study, Journal of Software Maintenance and Evolution: Research and Practice 22 (3) (2010) 165–189.

[21] S. J. Clarke, P. Willett, Estimating the recall performance of web search engines, in: Aslib proceedings, MCB UP Ltd, 1997.

[22] N. S. Junior, L. Rocha, L. A. Martins, I. Machado, A survey on test practitioners' awareness of test smells, arXiv preprint arXiv:2003.05613 (2020).

[23] N. Tsantalis, D. Mazinanian, G. P. Krishnan, Assessing the refactorability of software clones, IEEE Transactions on Software Engineering 41 (11) (2015) 1055–1090.

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

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

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

[27] L. Jiang, 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.

[28] M. White, M. Tufano, C. Vendome, D. Poshyvanyk, Deep learning code fragments for code clone detection, in: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2016, pp. 87–98.

[29] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, C. V. Lopes, Oreo: Detection of clones in the twilight zone, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 354–365.

[30] L. Jiang, G. Misherghi, Z. Su, 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.

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

[32] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes, Sourcerercc: Scaling code clone detection to big-code, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 1157–1168.

[33] J. Svajlenko, C. K. Roy, Cloneworks: A fast and flexible large-scale near-miss clone detection tool, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 177–179.

[34] B. van Bladel, 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.

[35] C. K. Roy, 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.

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

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

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

[39] P. Jaccard, Nouvelles recherches sur la distribution florale, Bull. Soc. Vaud. Sci. Nat. 44 (1908) 223–270.

[40] S.-S. Choi, S.-H. Cha, C. C. Tappert, A survey of binary similarity and distance measures, Journal of Systemics, Cybernetics and Informatics 8 (1) (2010) 43–48.

[41] C. J. Kapser, M. W. Godfrey, "cloning considered harmful" considered harmful: patterns of cloning in software, Empirical Software Engineering 13 (6) (2008) 645.

[42] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Transactions on software engineering 33 (9) (2007) 577–591.

[43] C. K. Roy, J. R. Cordy, A survey on software clone detection research, Queen's School of Computing TR 541 (115) (2007) 64–68.

[44] M. Kim, V. Sazawal, D. Notkin, G. Murphy, An empirical study of code clone genealogies, in: Proceedings ESEC/FSE 2005 (10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering), ACM, New York, NY, USA, 2005, p. 187–196. doi:10.1145/1081706.1081737.

[45] J. Krinke, A study of consistent and inconsistent changes to code clones, in: 14th Working Conference on Reverse Engineering (WCRE 2007), 2007, pp. 170–178.

[46] S. Kawuma, J. Businge, E. Bainomugisha, Can we find stable alternatives for unstable eclipse interfaces?, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), 2016, pp. 1–10.

[47] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, B. Baudry, A snowballing literature study on test amplification, Journal of Systems and Software 157 (2019).

[48] B. Danglot, O. L. Vera-Pérez, B. Baudry, M. Monperrus, Automatic test improvement with dspot: a study with ten mature open-source projects, Empirical Software Engineering, Springer Verlag (2019).

[49] M. Abdi, H. Rocha, S. Demeyer, Test amplification in the pharo smalltalk ecosystem, in: Proceedings IWST 2019 (International Workshop on Smalltalk Technologies), ESUG, 2019.

[50] M. Pyl, B. van Bladel, S. Demeyer, An empirical study on accidental cross-project code clones, in: 2020 IEEE 14th International Workshop on Software Clones (IWSC), IEEE, 2020, pp. 33–37.

[51] E. T. Barr, M. Harman, Y. Jia, A. Marginean, J. Petke, Automated software transplantation, in: Proceedings ISSTA 2015 (International Symposium on Software Testing and Analysis), ACM, New York, NY, USA, 2015, p. 257–269. doi:10.1145/2771783.2771796.

[52] T. Zhang, M. Kim, Automated transplantation and differential testing for clones, in: Proceedings ICSE 2017 (IEEE/ACM 39th International Conference on Software Engineering), 2017, pp. 665–676.

[53] C. K. Roy, M. F. Zibran, R. Koschke, The vision of software clone management: Past, present, and future (keynote paper), in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 18–33.

[54] E. Duala-Ekoko, M. P. Robillard, Clonetracker: Tool support for code clone management, in: Proceedings ICSE 2008 (30th International Conference on Software Engineering), ICSE '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 843–846. doi:10.1145/1368088.1368218.

[55] G. Zhang, X. Peng, Z. Xing, Shihai Jiang, Hai Wang, W. Zhao, Towards contextual and on-demand code clone management by continuous monitoring, in: Proceedings ASE 2013 (28th IEEE/ACM International Conference on Automated Software Engineering), 2013, pp. 497–507.

[56] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, T. Sano, Applying clone change notification system into an industrial development process, in: Proceedings ICPC (21st International Conference on Program Comprehension), 2013, pp. 199–206.

[57] S. Tokui, N. Yoshida, E. Choi, K. Inoue, Clone notifier: Developing and improving the system to notify changes of code clones, in: Proceedings SANER 2020 (IEEE 27th International Conference on Software Analysis, Evolution and Reengineering), 2020, pp. 642–646.