Test Code:
a New Frontier in Code Cloning Research

Proefschrift voorgelegd tot het behalen van de
graad van doctor in de wetenschappen: informatica
aan de Universiteit Antwerpen te verdedigen door:

Brent van Bladel

Promotor
prof. dr. Serge Demeyer

Faculteit Wetenschappen
Departement Informatica

Antwerpen, 2023

Universiteit Antwerpen

Test Code: a New Frontier in Code Cloning Research

Brent van Bladel

Promotor:

prof. dr. Serge Demeyer

Proefschrift ingediend tot het behalen van de graad van
doctor in de wetenschappen: informatica.

Promoter:

prof. dr. Serge Demeyer

Doctoral Jury:

prof. dr. Sigrid Eldh	Mälardalen University, Sweden
prof. dr. Rainer Koschke	University of Bremen, Germany
prof. dr. Moharram Challenger	University of Antwerp, Belgium
prof. dr. Guillermo Alberto Perez	University of Antwerp, Belgium
prof. dr. Serge Demeyer	University of Antwerp, Belgium

Acknowledgments

First of all, I would like to thank everyone who helped me realise this thesis. Above all, I would like to thank my supervisor, Serge Demeyer, for believing in me ever since my first research internship, for giving me the opportunity to work on the Cha-Q project, for teaching me almost everything I know about science and research, and of course for his continuous guidance throughout my PhD, without which this thesis would have never been possible. I would also like to thank Alessandro Murgia for taking me under his wings when I first started, for helping and guiding me with my very first papers, and for teaching me the difference between "counting" and "analysis". I would like to thank my colleagues in LORE, in particular Ali Parsai, Diana Leyva Pernia, Gusher Laghari, Sten Vercammen, Henrique Rocha, Mercy Njima, and John Businge, for helping me on many occasions. And I would like to thank Sigrid Eldh, Rainer Koschke, Guillermo Perez, and Moharram Challenger who accepted to be on the jury despite their busy schedule.

Secondly, I would like to thank everyone who made me enjoy going to the campus. I would like to thank all my colleagues I had the pleasure of teaching with side-by-side: Serge Demeyer, Simon Van Mierlo, Hans Vangheluwe, Stephen Pauwels, Guillermo Perez, Dirk Janssens, Jeroen Famaey, and Tom Hofkens. I would also like to thank all other colleagues who I have worked with, but who are too plentiful to all name. And I would like to thank my good friends Matthias Verstappen, Christophe Segers, and Robbe Van Gastel for all the fun moments we had on campus.

Finally, I would like to thank everyone who supported me throughout the past six years as I worked on this thesis. I would like to thank Isabelle Vloeberghs for all the help and support she provided when I needed it the most. I would like to thank Jitse Laenen for standing by my side throughout this whole adventure. And I would like to thank all my friends and family for the continuous support.
Thank you.

Brent van Bladel
Antwerp, Belgium, August 2022

Abstract

As software has become ever important in our lives, all that code needs to be of a high quality. A common way to achieve this is via software testing, where additional "test code" is written with the sole purpose of finding mistakes in the original code, or "production code". As test code has the large responsibility of ensuring qualitative software, it is critical that the test code itself is of high quality as well. However, while the quality of test code is often synonymous with its ability to find bugs, it is equally important to ensure readability and maintainability of test code to allow agile teams working incrementally to update, extend, and maintain the test code each iteration.

The presence of code duplication, or "code clones", can affect the readability and maintainability of code. While code clones have already been extensively researched in production code, research on test code duplication is limited. And yet, duplicate tests are a common occurrence, as the quickest way for a developer to test a new feature is to copy, paste, and modify an existing test. In this thesis, we address this gap in the literature by answering two research questions. First, we investigate whether the structure of test code can be exploited to detect semantic code clones. Second, we investigate whether test code duplication should be considered independently of production code duplication. In the end, we show that test code is a rich source for studying clones and that further investigation is warranted.

Nederlandstalige Samenvatting

Omdat software steeds belangrijker wordt in ons leven, moeten we een hoge kwaliteit van code nastreven. Dit wordt typisch gedaan aan de hand van softwaretesten, waarbij aanvullende "testcode" wordt geschreven met als doel fouten in de originele code, of "productiecode", te vinden. Aangezien testcode de verantwoordelijkheid heeft om de kwaliteit van software te garanderen, is het van cruciaal belang dat de testcode zelf ook van hoge kwaliteit is. En ondanks dat de kwaliteit van de testcode vaak synoniem staat met het vermogen om *bugs* te vinden, is het net zo belangrijk om de leesbaarheid en onderhoudbaarheid van de testcode te waarborgen, zodat *agile teams* de testcode incrementeel kunnen uitbreiden en onderhouden.

Duplicate code, of een zogenaamde "*code clone*", kan de leesbaarheid en onderhoudbaarheid van code beïnvloeden. Hoewel *code clones* reeds uitgebreid onderzocht werden in productiesystemen, is onderzoek naar duplicatie in testcode nog zeer beperkt. Desondanks komen duplicate testen veel voor, aangezien een ontwikkelaar het snelst een nieuwe functie kan testen door een bestaande test te kopiëren, plakken en wijzigen. In dit proefschrift pakken we dit gebrek in de literatuur aan door twee onderzoeksvragen te beantwoorden. Eerst onderzoeken we of de structuur van testcode gebruikt kan worden om semantische *code clones* te detecteren. Ten tweede onderzoeken we of testcodeduplicatie en duplicatie in productiesystemen afzonderlijk moeten beschouwen. Uiteindelijk tonen we aan dat testcode een rijke bron is voor het bestuderen van *code clones* en dat verder onderzoek gerechtvaardigd is.

Publications and Presentations

Papers included in this thesis.

Some papers have been slightly changed in this thesis according to feedback from the doctoral jury.

1. Brent van Bladel and Serge Demeyer. **Test Refactoring: a Research Agenda**. In *Post-proceedings of the Tenth Seminar on Advanced Techniques and Tools for Software Evolution(SATTOSE)*, 1–6. Madrid, Spain. June, 2017.

2. Brent van Bladel and Serge Demeyer. **Test Behaviour Detection as a Test Refactoring Safety**. In *Proceedings of the 2nd International Workshop on Refactoring(IWOR)*, 22–25. Montpellier, France. September, 2018.

3. Brent van Bladel and Serge Demeyer. **A Novel Approach for Detecting Type-IV Clones in Test Code**. In *Proceedings of the 13th International Workshop on Software Clones(IWSC)*, 8–12. Hangzhou, China. Februari, 2019.

4. Brent van Bladel and Serge Demeyer. **A Comparative Study of Test Code Clones and Production Code Clones**. In *Journal of Systems and Software(JSS)*, 110940. June, 2021.

5. Brent van Bladel and Serge Demeyer. **A Comparitative Study of Clone Evolution in Test Code and Production Code**. In *Proceedings of the 6th Workshop on Validation, Analysis and Evolution of Software Tests(VST)*. Macao, China. March, 2023.

Other papers published during the course of the PhD.

Paper (1) is not included in the thesis due to significant overlap with Chapter 7. Paper (2) is not included in the thesis as Chapter 6 is an extension of this paper. Papers (3), (4), and (5) are not included in the thesis as they fall outside the scope of our problem domain.

1. Brent van Bladel and Alessandro Murgia and Serge Demeyer. **An Empirical Study of Clone Density Evolution and Developer Cloning Tendency**. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering(SANER)*, 551–552. Klagenfurt, Austria. Februari, 2017.

2. Brent van Bladel and Serge Demeyer. **Clone Detection in Test Code: an Empirical Evaluation**. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering(SANER)*, 492–500. London, Canada. Februari, 2020.

3. Andrés Carrasco and Brent van Bladel and Serge Demeyer. **Migrating towards Microservices: Migration and Architecture Smells**. In *Proceedings of the 2nd International Workshop on Refactoring(IWOR)*, 1–6. Montpellier, France. September, 2018.

4. Mitchel Pyl and Brent van Bladel and Serge Demeyer. **An Empirical Study on Accidental Cross-project Code Clones**. In *Proceedings of the 14th International Workshop on Software Clones(IWSC)*, 33–37. London, Canada. Februari, 2020.

5. Serge Demeyer and Ali Parsai and Sten Vercammen and Brent van Bladel and Mehrdad Abdi. **Formal Verification of Developer Tests: a Research Agenda Inspired by Mutation Testing**. In *International Symposium on Leveraging Applications of Formal Methods(ISOLA)*, 9–24. Rhodes, Greece. October, 2020.

Presentations and tutorials given in the course of doctoral study:

1. 2017-02-23 **An Empirical Study of Clone Density Evolution and Developer Cloning Tendency** at SANER (Klagenfurt, Austria)
2. 2017-06-08 **Test Refactoring: a Research Agenda** at SATTOSE (Madrid, Spain)
3. 2018-09-04 **Migrating towards microservices: migration and architecture smells** at IWOR (Montpellier, France)
4. 2018-09-04 **Test behaviour detection as a test refactoring safety** at IWOR (Montpellier, France)
5. 2018-11-05 **Goal-oriented Mutation Testing with Focal Methods** at A-Test (Orlando, Florida)
6. 2018-11-05 **Mutation Testing, Hands-on Session** at A-Test (Orlando, Florida)
7. 2019-02-24 **A Novel Approach for Detecting Type-IV Clones in Test Code** at IWSC (Hangzhou, China)
8. 2020-02-18 **An Empirical Study on Accidental Cross-Project Code Clones** at IWSC (London, Canada)
9. 2020-02-21 **Clone Detection in Test Code: An Empirical Evaluation** at SANER (London, Canada)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software has taken over the world. It flies our planes, helps drive our cars, and makes our trains run on time. Hospitals and critical infrastructure, such as emergency services and the electric grid, all depend on software to operate effectively and efficiently. It would be difficult to find a single person whose life is not influenced by software in some way, as personal experience has shown me that even the most primitive Maasai tribes that roam the savannahs in Tanzania use cellphones to communicate.

As software has become ever important in our lives, all that code needs to be of a high quality. A common way to achieve this is via software testing. Simply put, another piece of software, called "test code", is written with the sole purpose of finding mistakes in the original code, or "production code".

In most organizations, the test code is the final "quality gate" for an application, allowing or denying the move from development to release. With this role comes a large responsibility: the success of an application, and possibly the organization, rests on the quality of the software product [1, 2]. Therefore, it is critical that the test code itself is of high quality.

The quality of test code is often synonymous with its ability to find bugs. Methods such as code coverage analysis and mutation testing help developers assess the effectiveness of their test code, and are therefore commonly used as metrics to measure quality in the test suite. The quickest way for a developer to improve these metrics is to extend existing unit tests or add new unit tests, regardless of how this affects the test code itself.

However, with agile teams working incrementally on production code, the test code needs to be updated, extended, and maintained each iteration as well. As a result, it is equally important to ensure quality of the test code in terms of readability and maintainability, as the test code should allow for these repeated changes to be easily applied. It has become a recommended practice to continuously monitor the readability and maintainability of the test suite [3, 4].

The presence of code duplication, or "code clones", can affect the readability and maintainability of code. In test code, duplicate tests are a common occurrence, as the quickest way for a developer to test a new feature is to copy, paste, and modify an existing test [5]. Even if a developer does create a new test from scratch, the consistent structure of unit test code can still cause clones accidentally. Yet, while code clones have already been extensively researched in production code, research on test code duplication is limited [6].

In this thesis, we address this gap in the literature by investigating duplication in test code. Our investigation focusses specifically on unit and integration test code, yet in the interest of brevity, we will simply use the term "test code" in the rest of this thesis. We try to address the gap in the literature by answering two main research questions, and in the end, we show that test code is a rich source for studying clones and that further investigation is warranted.

RQ1: *Can we exploit the structure of test code to detect semantic code clones?*
While many tools already exists to detect syntactically similar code fragments, detecting semantical similarity remains a difficult task. We propose that typical test idioms and the consistent structure of test code can be exploited to find semantically similar tests. This would enable both practitioners and researchers to find a large set of interesting test clones, allowing better refactoring efforts and further in-depth studies for practitioners and researchers respectivly. We will address this research question in Chapters 3 to 5 by implementing a semantic test clone detection tool.

RQ2: *Should test code duplication be considered independently of production code duplication?*
Since test code differs significantly from production code in its structure, we propose that duplication in test code also differs significantly from duplication in production code. This would mean that current clone detection, refactoring, and maintainance approaches might not be optimised for test code, and that current code cloning research might not apply for test code. We will address this research question in Chapters 6 to 7 by performing a series of empirical studies into test code duplication, comparing it with production code duplication.

1.1 CONTRIBUTIONS

The main contributions of this thesis are as follows.

- In Chapter 3, we provide a theoretical model that defines the behaviour of a test case. This model provides a basis that can be used to detect semantic clones in test code.
- In Chapter 4, we show that it is feasible to detect test behaviour with our theoretical model. We do this by creating T-CORE, a tool that uses our model to detect changes in test behaviour.
- In Chapter 5, we show that it is feasible to detect semantic clones in test code using our theoretical model. We do this by using T-CORE to detect 259 semantic clones in a large open-source project.
- In Chapter 6, we perform a comparative study between code clones in test code and in production code. We find that test code contains more than twice as much duplication compared to production code, that clones in production code cause a significant increase in test clones, and that clones in test code inherently differ from clones in production code.
- In Chapter 7, we perform a larger comparative study that focusses on clone evolution in test code and in production code. We conclude that the amount of duplication in test code is significantly higher than in production at every point during development, that clone density in test code much more sensitive is to changes in the codebase compared to production code, and that clones in test code are being changed slightly more often than in production code, yet they are more likely to be changed consistently than inconsistently.

1.2 ORIGINS OF CHAPTERS

Chapters 3 to 6 are peer-reviewed and published. Chapter 7 will be published in March 2023.

Chapter 3 was published in the *Post-proceedings of the Tenth Seminar on Advanced Techniques and Tools for Software Evolution* (*SATTOSE*) [7].

Chapter 4 was published in the *Proceedings of the 2nd International Workshop on Refactoring* (*IWOR*) [8].

Chapter 5 was published in the *Proceedings of the 13th International Workshop on Software Clones* (*IWSC*) [9].

Chapter 6 was published in the *Journal of Systems and Software* (*JSS*) [10] as an extension of our previously published paper in the *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering* (*SANER*) [11].

Chapter 7 will be published in the *Proceedings of the 6th Workshop on Validation, Analysis and Evolution of Software Tests* (*VST*).

Chapter 2

Terminology

In this chapter, we define the terminology necessary to understand the rest of this thesis.

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. One 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 density. Clone density, also known as *clone percentage* [12], or the *Total Cloned Method percentage* (TCMp) or the *Total Cloned Line of Code percentage* (TCLOCp), depending on the granularity [13, 14], is a metric that describes the amount of duplication in a codebase. For a function-level granularity, it 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 that appear in at least one clone fragment. In this thesis, we always refer to clone density at a function-level granularity.

Clone type. 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 clones are exactly the same, only allowing differences in comments, whitespaces, and indentation.

— Type-2 clones are type-1 clones that also have differences in variable names and literal values.

— Type-3 clones are type-2 clones that also contain added or removed lines of code.

Semantic clones. on the other hand are code clones that are semantically similar without necessarily being syntactically similar. These are often called type-4 clones.

Clone lineage. A clone lineage is the ordered set of all versions of one clone class throughout development. Each version corresponds to the clone class at a certain commit of the project, from its introduction to its removal or to the last commit of the project.

Clone genealogy. A clone genealogy is a set of clone lineages that have originated from the same clone. This occurs when a subset of the clone fragments in a clone class are changed, such that these clone fragments are still syntactically similar with each other, but no longer syntactically similar to the other clone fragments from the original clone class. As a result, the subset of clone fragments will form a new clone class by itself, and the original clone class now consists of a smaller set of clone fragments, yet both originate from the same clone.

Part I

Semantic Clone Detection in Test Code

Chapter 3

Defining Test Code Semantics

Test Refactoring: a Research Agenda

Brent van Bladel and Serge Demeyer
In *Post-proceedings of the Tenth Seminar on Advanced Techniques and Tools for Software Evolution (SATTOSE)*, 1–6. Madrid, Spain. June, 2017.

This chapter was originally published in the *Post-proceedings of the Tenth Seminar on Advanced Techniques and Tools for Software Evolution (SATTOSE)*.

CONTEXT
In this chapter, we provide a theoretical model that defines the behaviour of a test case. This model will provide the basis of our semantic test clone detector in an attempt to answer our first main research question. Note that this chapter presents our theoretical model as part of a larger proposed research agenda on test quality. Part of this research plan is out of scope for this thesis, so we leave this as future research.

ABSTRACT *Research on software testing generally focusses on the effectiveness of test suites to detect bugs. The quality of the test code in terms of maintainability remains mostly ignored. However, just like production code, test code can suffer from code smells that imply refactoring opportunities. In this paper, we will summerize the state-of-the-art in the field of test refactoring. We will show that there is a gap in the tool support, and propose future work which will aim to fill this gap.*

3.1 INTRODUCTION

Refactoring is "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [15]. If applied correctly, refactoring improves the design of software, makes software easier to understand, helps to find faults, and helps to develop a program faster [15].

In most organizations, the test code is the final "quality gate" for an application, allowing or denying the move from development to release. With this role comes a large responsibility: the success of an application, and possibly the organization, rests on the quality of the software product [1]. Therefore, it is critical that the test code itself is of high quality. Methods, such as code coverage analysis and mutation testing, help developers assess the effectiveness of the tests suite. Yet, there is no metric or method to measure the quality of the test code in terms of readability and maintainability.

One indication of the quality of test code could be the presence of test smells. Similar to how production code can suffer from code smells, these test specific smells can indicate problems with the test code in terms of maintainability [16]. However, refactoring test smells can be tricky, as there is no reliable method to verify if a refactored test suite preserves its external behaviour. Several studies point out the peculiarities of test code refactoring [15, 16, 17, 18]. However, none of them provided an operative method to guarantee that such refactoring was preserving the behaviour of the test.

The rest of the paper is organized as follows. In section 3.2 we will summerize the related work on test smells and test refactoring, which shows test smells to be an important issue. Section 3.3 we will go over the existing test refactoring tools, showing there is a gap in the current tool support. We will propose our future work which aims to fill the gap in existing tool support in section 3.4. In section 3.5 we define a theoretical model for defining test behaviour, which will form the basis of our proposed future work. We conlude in section 3.6.

3.2 RELATED WORK

The term test smell was first introduced by van Deursen et al. in 2001 as a name for any symptom in the test code of a program that possibly indicates a deeper problem. In their paper, they defined a first set of eleven common test smells and a set of specific refactorings which solve those smells [16]. Meszaros expanded the list of test smells in 2007, making a further distinction between test smells, behaviour smells, and project smells [19]. Greiler et al. defined five new test smells specifically related to test fixtures in 2013 [20].

Several studies have investigated the impact test smells have on the quality of the code. Van Rompaey et al. performed a case study in 2006 in which they investigated two test smells (*General Fixture* and *Eager Test*). They concluded that all tests which suffer from these smells have a negative effect on the maintainability of the system [21]. In 2012, Bavota et al. performed an experiment with master students in which they studied eight test smells (*Mystery Guest*, *General Fixture*, *Eager Test*, *Lazy Test*, *Assertion Roulette*, *Indirect Testing*, *Sensitive Equality*, and *Test Code Duplication*). This study provided the first empirical evidence of the negative impact test smells have on maintainability [22]. In 2015, they continued their research and performed the experiment with a larger group, containing more students as well as developers from industry. They conclude that test smells represent a potential danger to the maintainability of production code and test suites [23].

In 2016, Tufano et al. investigated the nature of test smells. They conducted a large-scale empirical study over the commit history of 152 open source projects. They found that test smells affect the project since their creation and that they have a very high survivability. This shows the importance of identifying test smells early, preferably in the IDE before the commit. They also performed a survey with 19 developers which looked into their perception of test smells and design issues. They showed that developers are not able to identify the presence of test smells in their code, nor do developers perceive them as actual design problems. This highlights the importance of investing effort in the development of tools to identify and refactor test smells [24].

3.3 TOOL SUPPORT

3.3.1 Test Smell Detection

There are many tools that can automatically detect code smells, for example the JDeodorant Eclipse plugin and the inFusion tool [25]. Test smells, however, are very different from code smells and these tools are not able to detect them. Tool support for handling test smells and refactoring test code is limited.

In 2008, Breugelmans et al. presented TestQ, a tool which can statically detect and visualize 12 test smells [26]. TestQ enables developers to quickly identify test smell hot spots, indicating which tests need refactoring. However, the lack of integration in development environments and the overall slow performance make TestQ unlikely to be useful in rapid code-test-refactor cycles [26].

In 2013, Greiler et al. presented a tool which can automatically detect test smells in test fixtures [20]. A test fixture is a set of fixed steps that setup and teardown the object under test in a consistent way for multiple unit tests. Their tool, called TestHound, provides reports on test smells and recommendations for refactoring the smelly test fixtures. They performed a case study where developers are asked to use the tool and afterwards are interviewed. They show that developers find that the tool helps them to understand, reflect on and adjust test code. However, their tool is limited to smells related to test fixtures. Furthermore, they only report the occurences of the different fixture-related test smells in the code. They do not give one single metric that represents the overall quality of the test code. During the interviews, one developer said that the different smells should be integrated in one high-level metric: "This would give us an overall assessment, so that if you make some improvements you should see it in the metric." [20].

3.3.2 Defining Test Behaviour

Refactoring of the production code can be done with little risk using the test suite as a safeguard. However, since there is no safeguard when refactoring test code, there is a need for tool support that can verify if a refactored test suite preserves its behaviour pre- and post-refactoring. Previous research on this topic has been performed by Parsai et al. in 2015 [27]. They propose the use of mutation testing to verify the test behaviour. However, mutation testing requires the test suite to be ran for each mutant, which can be hundreds of times, making it unlikely to be useful in practice.

As an alternative, one could consider line coverage or branch coverage, as this would only require the test suite to be ran once. However, while this can provide an indication of the test behaviour, it cannot fully guarantee that the test behaviour is preserved. For example, changing the value of a test parameter changes the test behaviour while resulting in the same line coverage. As such, there is still room for improvements when it comes to verifying test behaviour.

3.4 RESEARCH PLAN

As we have shown, there is a lack of tool support when it comes to test refactoring. We plan on creating a tool that will help developers during this process. We present our future work in terms of a research agenda:

3.4.1 Test Smell Detection

— *Objective* - Create a tool that is able to detect test smells. More specifically, the tool should be able to detect all test smells defined by van Deursen, Meszaros, and Greiler [16, 19, 20]. This tool should also be able to create a metric that represents the overall quality of the test code in terms of maintainability.

— *Approach* - Breugelmans et al. proposed methods for detecting all the original test smells (defined by van Deursen et al.) [26]. We will use these methods in our tool. For the other test smells (defined by Meszaros and Greiler et al.), we will use a similar approach in order to define detection methods ourselves. The metric that represents the overall quality of the test code can be calculated based on the number of test smells present in the test code.

— *Validation* - Verification of correctness will be made using a dataset consisting of a set of real open-source software projects. We can compare the tool with TestHound for fixture related test smells and with TestQ for the other test smells. Smells not covered by either TestHound or TestQ will require manual verification.

3.4.2 Defining Test Behaviour

— *Objective* - Define test behaviour such that developers can verify if the test code is behaviour preserving between pre- and post- refactoring.

— *Approach* - The production code should be deterministic, and thus the same set of inputs should always result in the same set of outputs. We will analyse the code in order to map all entry and exit points from test code to production code and link them with the corresponding assertions. This will result in the construction of a Test Behaviour Tree (TBT), which defines the behaviour of the test. Comparison of TBTs will allow for validating behavior preservation between pre- and post- refactoring. Section 3.5 will explain this concept in more detail.

— *Validation* - We will run the algorithm on the dataset of commits used for verifying the test quality metric. We can do an initial check using coverage metrics and mutation testing. When these metrics change pre- and post-refactoring, we know for certain that the test behaviour changed. When these metrics remain constant, we will have to manually verify wether the refactoring is behaviour preserving.

3.5 THEORETICAL MODEL FOR DEFINING TEST BEHAVIOUR

In order to determine test behaviour, we propose that a Test Behaviour Tree (TBT) can be constructed from the Abstract Syntax Tree (AST) of the test. To construct the TBT, start from the AST of the unit test and choose the scope-node of the unit test as the root for the TBT. This will result in a tree that has a child for every statement in the unit test. Then, for every statement that is not an assert statement, remove the corresponding node and its subtree from the TBT. This results in a TBT where the root-node has a child-node for every assert statement.

Then we replace all variable-nodes in the TBT with a literal-node containing their value at runtime. If a variable is initialized with a functioncall to production code, it should be represented as such. Since the implementation of production code does not affect the behaviour of the test code, we consider it as a 'black box'. Further operations on that variable will then be stored as the sequence of those operations on that object. The final tree represents what is being tested, and thus describes the behaviour of the test.

3.5.1 Running Example

As an example to illustrates the approach, we use the following simple production code:

```
class Rectangle{
public:
   Rectangle();
   int getHeigth();
   int getWidth();
   void setHeigth(int h);
   void setWidth(int w);
private:
    int heigth;
    int width;
};

Rectangle::Rectangle(){}
int Rectangle::getHeigth() { return heigth; };
int Rectangle::getWidth() { return width; };
void Rectangle::setHeigth(int h) {heigth = h;}
void Rectangle::setWidth(int w) {width = w;}
```

It defines a class `Rectangle` which has two private data members `heigth` and `width`, as well as getters and setters for these data members. Note that even though this is a toy example, there is no technical difference between simple getters and setters and large algoritmic functions as the production code is considered a 'black box'. There would be no difference if the getters did some advanced mathematical calculations, read from a file, or contacted a networked database.

We will start with a simple test for this production code:

```
Rectangle r = Rectangle();
r.setWidth(5);
r.setHeigth(10);
assert(5 == r.getWidth());
assert(10 == r.getHeigth());
```

This test will result in the Test Behaviour Tree shown in figure 3.1. As shown, the TBT has one root node which has a child for every assert. Each assert node has the full comparison as a child, where variables are replaced with their value. Since the call on the rectangle object is considered a call to production code, the sequence of operations is appended as a child rather than a single value, because we consider production code as a 'black box'. We can safely assume this, since the production code should be deterministic (otherwise you could not write tests for it) and should not change when refactoring test code.

3.5.2 Variable Refactorings

One way to refactor this test would be to replace the 'magic numbers' in the with variables. This would greatly increase maintainability, as consistency between input and expected output would be guaranteed. Because variables are replaced with their value in our approach, the following refactored test code will result in the exact same TBT:

```
int x = 5;
int y = 10;
Rectangle r = Rectangle();
r.setWidth(x);
r.setHeigth(y);
assert(x == r.getWidth());
assert(y == r.getHeigth());
```

Similarly, the common refactoring where a variable is renamed can be performed without changing the TBT. The following code also generates the same TBT.

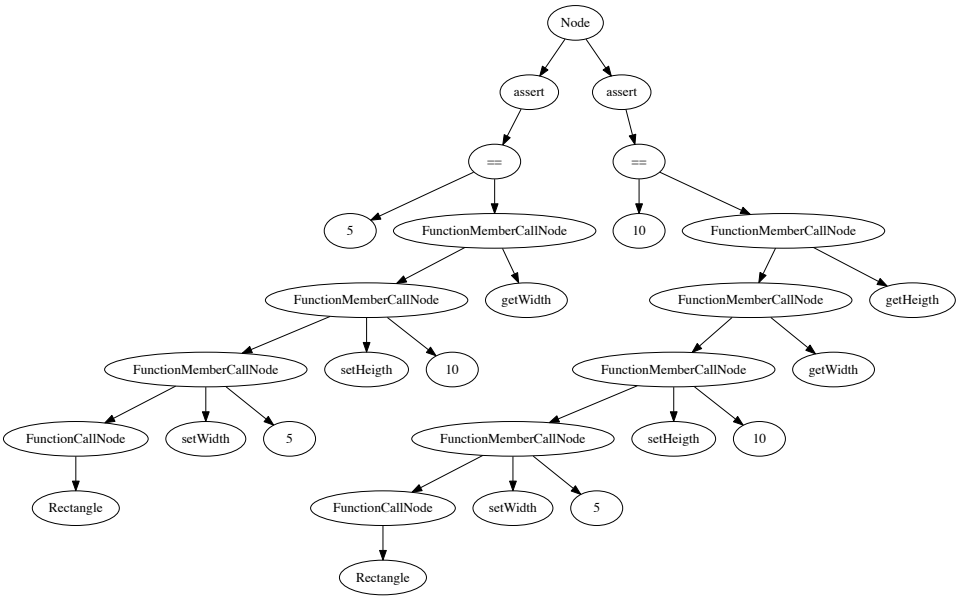

Figure 3.1: The Test Behaviour Tree from the example tests.

```
int testWidth = 5;
int testHeigth = 10;
Rectangle testRectangle = Rectangle();
testRectangle.setWidth(testWidth);
testRectangle.setHeigth(testHeigth);
assert(testWidth == testRectangle.getWidth());
assert(testHeigth == testRectangle.getHeigth());
```

These refactorings did not change behaviour, which is why we get the same resulting TBT. If you would change the value of testWidth or testHeigth, the behaviour of the test would change as you would be testing different input - output pairs. This change in behaviour would be detected easily detected by our approach, as the values in the TBT would change accordingly, resulting in a different TBT.

3.5.3 Expression Refactorings

Detecting a change in input - output pairs is more important when the test code contains some arithmetic operations. Sometimes it is necessary to make a calculation in the test code to use as an oracle. When it comes to these kind of expressions in the AST, it is

possible to simply evaluate them during traversal of the AST. The values of all variables are stored upto that point in the program, and the result can be stored as the new value for the corresponding variable. Therefore, the following code still generates the same TBT, as the behaviour did not change since the values for testWidth and testHeigth still evaluate to 5 and 10 respectively [1]):

```
int testWidth = 1;
int testHeigth = ((++testWidth) * 2) + ((testWidth++) * 3) + 2;
testWidth = testWidth++;
Rectangle testRectangle = Rectangle();
testRectangle.setWidth(testWidth);
testRectangle.setHeigth(testHeigth);
assert(testWidth == testRectangle.getWidth());
assert(testHeigth == testRectangle.getHeigth());
```

3.5.4 Function Refactorings

Another common refactoring is to extract part of the test code to a function. As an example, we could define the following functions:

```
int setupWidth(int x){
    return x/2;
}

int setupHeigth(int y){
    return y*2;
}
```

and rewrite our test to:

```
int testWidth = setupWidth(10);
int testHeigth = setupHeigth(5);
Rectangle testRectangle = Rectangle();
testRectangle.setWidth(testWidth);
testRectangle.setHeigth(testHeigth);
assert(testWidth == testRectangle.getWidth());
assert(testHeigth == testRectangle.getHeigth());
```

If these functions are marked as part of the production code, they will be treated as 'black box' functions. This is not desirable, since then the TBT will change while behaviour is preserved. Therefore, these functions need to be evaluated similarly to expres-

[1] Note that it would be bad practice to write this test, but we use it here simply to showcase the approach.

sions. Again this is perfectly possible since we have the values of all variables at each point in the program. Upon evaluation, the values for testWidth and testHeigth still result in 5 and 10 respectively, and thus the TBT would be unchanged.

3.5.5 Conditionals and Loops

Upto now, our examples did not contain any conditionals or loops, since they are not desirable in test code. However, sometimes they could appear in test code, in which case they can be evaluated similarly to expressions and function calls. For example, we could define the following function:

```
int setupData(int i){
    if (i == 1){
        return 5;
    } else {
        if (i == 2) {
            return 5 + 5;
        }
    }
    return 0;
}
```

and rewrite our test to:

```
int testWidth = setupData(1);
int testHeigth = setupData(2);
Rectangle testRectangle = Rectangle();
testRectangle.setWidth(testWidth);
testRectangle.setHeigth(testHeigth);
assert(testWidth == testRectangle.getWidth());
assert(testHeigth == testRectangle.getHeigth());
```

Again, the values for testWidth and testHeigth still evaluate to 5 and 10 respectively, resulting in the same TBT. When conditionals or loops are used in combination with calls to production code, it would be handled similarly to how the testRectangle object is handled. The sequence of operations would be kept, including the conditional or loop, similarly to how they would be represented in AST form.

3.6 CONCLUSION

We have presented an overview of the research done in the field of test smells and test refactoring. Research has indicated that test smells have a negative impact on maintainability and therefore need to be refactored. We have shown that there is a lack of tool support to aid developers with test refactoring. We also provided a theoretical model that defines test behaviour, in the form of Test Behaviour Trees, which can be used to compare test behaviour pre- and post-refactoring. We plan to create a tool for test refactoring which can detect test code smells, evaluate the test quality, and assure behaviour is preserved after test refactoring using our theoretical model. We currently have a working prototype for the latter. Our final tool will help developers decide when and where to refactor the test code, as well as help them perform the refactorings correctly, allowing developers to improve their test suite quickly and with confidence.

Chapter 4

Implementing a Tool to Capture Test Code Semantics

Test Behaviour Detection as a Test Refactoring Safety

Brent van Bladel and Serge Demeyer
In *Proceedings of the 2nd International Workshop on Refactoring* (*IWOR*), 22–25. Montpellier, France. September, 2018.

This chapter was originally published in the *Proceedings of the 2nd International Workshop on Refactoring* (*IWOR*).

CONTEXT
In this chapter, we present T-CORE, a tool that uses our theoretical model of test behaviour to detect changes in test semantics. While this version of the tool was not yet able to detect semantic clones, it does show that it is feasible to capture test behaviour using our theoretical model. As a use-case, we propose that this version of the tool can be used to verify that test semantics remain unchanged after the refactoring of test code.

ABSTRACT *When refactoring production code, software developers rely on an automated test suite as a safeguard. However, when refactoring the test suite itself, there is no such safeguard. Therefore, there is a need for tool support that can verify if a refactored test suite preserves its behaviour pre- and post-refactoring. In this work we present T-CORE (Test COde REfactoring tool); a tool that captures the behaviour of Java tests in the form of a Test Behaviour Tree. T-CORE allows developers to verify that the refactoring of a test suite has preserved the behaviour of the test.*

4.1 INTRODUCTION

In most organizations, the test suite is the final "quality gate" for an application, permitting or denying the move from development to release. With this role comes a large responsibility: the success of an application, and possibly the organization, rests on the quality of the software product [1]. Therefore, it is critical that the test code itself is of high quality.

Metrics such as code coverage [28] and techniques such as mutation testing [29] are used to measure how effective a test suite is in terms of catching bugs. Yet, the quality of a test suite is not only determined by its effectiveness. With software projects continuously evolving during development, it is important that the test code is as easily modifiable as the production code [16]. Non-functional requirements, such as readability and maintainability, are therefore also an important part of the quality of a test suit.

One indication of the quality of test code is the presence of test smells. Similar to how production code can suffer from code smells, these test specific smells can indicate problems with the test code in terms of maintainability [16].

Refactoring test code suffering from test smells can be tricky, as there is no reliable method to verify if a refactored test suite preserves its external behaviour. Several studies point out the peculiarities of test code refactoring [16, 17, 18]. However, none of them provided an operative method to guarantee that such refactoring was preserved the behaviour of the test. In this paper we present T-CORE, a tool that tackles this problem.

4.2 RELATED WORK

The term test smell was first introduced by van Deursen et al. in 2001 as a name for any symptom in the test code of a program that possibly indicates a deeper problem [16]. In their paper, they defined a first set of eleven common test smells and a set of specific refactorings that solve those smells. Since then, several studies have investigated the impact of test smells on the quality of the code. All these studies concluded that test smells have a negative impact on maintainability [21, 22, 23]. This shows the importance of refactoring test code.

While refactoring of the production code can be done with little risk using the test suite as a safeguard, there is no safeguard when refactoring the test code itself. There is a need for tool support that can verify if a refactored test suite preserves its behaviour pre- and post-refactoring. Previous research on this topic has been performed by Parsai

Figure 4.1: Example of a part of the T-CORE output when run on a behaviour changing refactor.

et al. in 2015 [27]. They propose the use of mutation testing to verify the test behaviour. However, mutation testing requires the execution of the complete test suite for each mutant, which makes it computationally expensive.

In 2017, Garousi et al. conducted a 'multivocal' literature mapping on both the scientific literature and also practitioners' grey literature, surveying all the sources on test smells and their refactoring [30]. In their work, they present an exhaustive list of tools for refactoring test code. All these tools focus on detecting test smells, while none of them help the developer actually perform the refactoring. With T-CORE, we fill this lack of tool support.

4.3 T-CORE

In this section we will provide a detailed overview of the implementation of our tool, as well as how it can be used.

4.3.1 Test Behaviour Detection

In order to capture test behaviour, we construct the *Test Behaviour Tree* [7]. This tree, which is similar to the *Abstract Syntax Tree*, represents the behaviour of the test code.

We construct the test behaviour tree in a way that is inspired by the technique named symbolic execution [31, 32]. During symbolic execution, one supplies symbols representing a set of possible values to the program instead of the normal inputs (e.g. numbers). The execution proceeds as normal except that values may be symbolic formulas over the input symbols [32]. For example: if a program takes an input `x`, then the value of a variable `var = x + 2` would be stored as the formula $x + 2$ after symbolic execution instead of an actual value.

We use this technique in combination with a special feature of unit test code, namely that most of the values of the variables can be deduced from the local code. This allows us to annotate the variables with concrete values in most situations and consequently can derive an accurate approximation of the test values. This technique is based on the constant folding optimization technique used in some compilers [33].

Practically, we construct the test behaviour tree by traversing the abstract syntax tree. During this pass of the abstract syntax tree, all variables and objects are stored, similar to how a compiler stores variables and objects in symbol table [34]. However, the difference with a compiler is that we also store the value of variables if possible, or the formula that represents the value otherwise. This way, all subsequent operations on variables can be performed with this stored value. If a function call to production code occurs in the test, the value will be a formula that includes this call. We can treat all calls to production code as "black box" operations because production code does not define test behaviour. Further operations will then result in formulas that include the calls to production code as a symbol rather than a value. When we encounter an assert in the test code, a node representing that assert is added to the test behaviour tree. This node will have one child nodes for the input and one for the expected output, for both of which we know the value or formula.

Once the test behaviour tree is constructed for both the pre- and post-refactoring test cases, we can compare them to determine if the test behaviour was preserved. Since variables have been replaced with their value or a symbol, and expressions have been reduced to their most simple form, both trees should be mostly identical in order for behaviour to be preserved.

4.3.2 Architecture

T-CORE is implemented in Python. We use ANTLR to parse Java test code and construct the abstract syntax tree [35]. The abstract syntax tree is then traversed in Python to create the test behaviour tree. Creation and comparison of the test behaviour trees can be done by running T-CORE via the command line, or by using the user interface.

Figure 4.2: Example of a part of the T-CORE trace information.

The user interface was built using flask [36]. Flask is a microframework for Python based on Werkzeug, which allows for the creation of web applications. Since Python and Flask are the only dependencies, T-CORE is lightweight and easily installed. It can be run locally for small development teams, or it can be run on a server for larger development teams. In the latter case, developers do not need to install anything on their machine: the T-CORE tool can then be accessed using any web browser.

4.4 USAGE SCENARIO

Prelude: Bob is a member of a small development team that writes and maintains safety-critical software for a controller in a large factory. Their system is extensively tested, and the test values were calculated by external domain experts. As the production code grew, so did the test code. However, while the production code was regularly refactored to improve maintainability, the test code was not due to time constraints. As long as a test was green, it was not touched. This has made it gradually more difficult to find the cause of failing tests, as tests were now incorrectly named or located. The team decided it was time for improvement, and Bob was appointed the task of refactoring the test code.

Scene 1: Bob has refactored the test code: tests were renamed, duplicated test code was extracted into methods, and assertions were moved to more appropriate test cases. He runs the tests, and everything is green. However, he wants to be sure that he did not make a mistake, as a missing assertion or a changed test value could allow critical bugs to go undetected in the future. He opens his favorite web browser and surfs to their server where T-CORE is hosted. He uploads the test files from before and after the refactoring. Once uploaded, all tests in each file are listed and the user is asked to select the test cases

that need to be checked. Since Bob wants to make sure nothing changed in the entire test suite, he simply selects all tests. Bob then presses the *compare* button and the tool shows the result.

T-CORE shows whether the behaviour was preserved or not with a big color-coded (green or red) message. Bob sees the red message pop up and realises he made a mistake. Figure 4.1 shows an example of this output. As shown in Figure 4.1, when the behaviour is changed, the percentage of asserts that are different is shown. Then, an overview of the test files from before and after the refactoring is shown. In this overview, each assert is marked green if it was preserved and red if it was changed. This allows Bob to quickly find the tests that where not refactored correctly.

After Bob has found the incorrectly refactored tests, he investigates what happened. He quickly realises that he initialised the object under test differently, which results in a different program state being tested. After a quick fix, he uploads the new test files to T-CORE and runs the comparison again. This time, Bob is happy to see the big green notification saying test behaviour was not changed.

Scene 2: A few weeks after Bob refactored the test suite, one of the tests fails. Alice, a colleague of Bob, recognizes the failing assertion as the same bug they fixed a few weeks earlier. Alice wants to lookup the commit that previously fixed this bug. However, since the assert was moved and the test cases were renamed, she has trouble finding the fixing commit. Therefore, she asks Bob for help. Bob opens up the T-CORE report he saved after refactoring the test suite, and goes to the detailed traceability overview. In this overview, each original assert is presented next to its refactored version, an example of which is shown in Figure 4.2. Bob looks up the failing assert, and immediately sees from which test case it came originally. This information helps Alice to find the fixing commit, allowing her to solve the bug.

4.5 VALIDATION

4.5.1 Benchmark

First, we tested our tool on an artificial benchmark we constructed to confirm that T-CORE behaves as expected for a base-line of known test refactorings. This benchmark is based on the following code.

```
int testWidth = 5;
int testHeight = 10;
Rectangle r = new Rectangle();
r.setWidth(testWidth);
r.setHeight(testHeight);
assertEquals(testWidth, r.getWidth());
assertEquals(testHeight, r.getHeight());
```

The benchmark consists of a set of 32 versions of this test code, where each version represents the same test before or after a test refactoring. These refactorings include the renaming of variables, the replacing of literal values with expressions, the extracting of these expressions in functions, the moving of initialisation to a setup method, and the removal of conditionals. In this set, 16 refactorings are behaviour preserving versions of the test code, while the other 16 refactorings change the behaviour. These latter refactorings were created by introducing a small bug or mistake during the refactoring that could also be made by the developer while performing the refactoring. We then ran our tool on each of these refactorings, and could easily verify whether the behaviour was correctly classified as either behaviour preserving or behaviour changing.

A tool that always outputs one class, or a tool that guesses randomly, would achieve an accuracy of 50% on this data set. This provides a base line to compare our tool with. When running T-CORE on this dataset, we get the results shown in Table 4.2 in the form of a confusion matrix. We can see that our tool achieves an accuracy of 100%, as it correctly classifies all 16 behaviour preserving refactorings correctly as well as all 16 behaviour changing refactorings. Therefore we can state that our tool is able to correctly detect whether the behaviour was preserved or changed in 100% of the cases.

4.5.2 Open-Source Projects

While our benchmark covers a lot of technical language features, we cannot claim that it covers everything from the Java language. Therefore, to increase our confidence that our tool works on a significant part of the Java language, we also tested it on a small set of open-source refactorings.

We use the Boa framework [37] to find instances of test refactorings in open-source projects from GitHub, and then we manually selected 5 commits such that each covered a different kind of refactoring or specific language feature.

Table 4.1 provides an overview of this set. Note that the last column shows the size of the refactored test, not the size of the project, since our tool does not analyze the entire project but only the relevant test code. After a manual analysis of these open-source refactorings, we found that all 5 preserved the behaviour of the test code. We then ran

Table 4.1: Open-soure test refactorings.

Project	**Refactoring**	**LOC**
Enero-String-Calculator	Extract data to test setup.	55
CodeKatas	Extract data to test setup.	61
soen490	Extract data to test class.	69
Beta	Variable and methods rename.	132
spring-data-simpledb	Extract duplicate code to method.	236

Table 4.2: Confusion matrix of the results of our tool.

	Tool Result	
	Behaviour Preserving	Behaviour Changing
Behaviour Preserving Refactoring	16	0
Behaviour Changing Refactoring	0	16

our tool on each of these refactorings, and observed that our tool correctly detected that the behaviour was preserved for all 5 refactorings. We conclude that our tool achieves an accuracy of 100% on this set of open-source projects as well. We also validated the correctness of the traceability report by manually verifying that all test cases matched where in fact the same test case.

4.6 CONCLUSION

In this work we presented T-CORE, a tool that infers the behaviour of test code, allowing developers to verify that the refactoring of a test suite preserved the behaviour of the test, as well as providing valuable traceability information. The traceability information provided by T-CORE allows the developer to trace an assert back through refactorings that changed the name of the test of moved it to another test.

Chapter 5

Using Test Code Semantics for Code Clone Detection

A Novel Approach for Detecting Type-IV Clones in Test Code

Brent van Bladel and Serge Demeyer
In *Proceedings of the 13th International Workshop on Software Clones (IWSC)*, 8–12. Hangzhou, China. Februari, 2019.

This chapter was originally published in the *Proceedings of the 13th International Workshop on Software Clones (IWSC)*.

CONTEXT
In this chapter, we provide an answer to our first main research question by adapting T-CORE to detect semantic clones in the test code of a large open-source project.

ABSTRACT *The typical structure of unit test code (setup - stimulate - verify - teardown) gives rise to duplicated test logic. Researchers have demonstrated the widespread use of syntactic clones in test code, yet if duplicated test code is indeed a problem, then semantic clones may be an issue as well. However, while detecting syntactic similarities can be done relatively easy, semantic similarities are more difficult to find. In this paper we present a novel way of detecting semantic clones by exploiting the unique features present in test code. We demonstrate on the Apache Commons Math Library's test suite that our approach can detect 259 semantic clones, of which only 54 were also detected by NiCad. This confirms that it is both feasible and worthwhile to investigate semantic clones in test code.*

5.1 INTRODUCTION

The introduction of the Agile methodology and extreme programming techniques had a major impact on how developers create software systems [38]. The focus shifted towards writing working increments of the software system, in contrast to the waterfall approach [39]. With each increment of the production code, the test code is updated, extended, and maintained as well. The quality of the test code quickly became as important as the quality of the production code. An intitial list of test smells was introduced in literature by Van Deursen et al. [16].

One of these initial test smells was *Test Code Duplication*, which is similar to the smell for production code [40]. However, the structure of unit test code is much more consistent than that of production code, with the idiomatic structure of the *setup-stimulate-verify-teardown* (S-S-V-T) cycle [41]. This causes duplication to be much more common in test code, with some systems having half of the entire test code being syntactic clones [42].

While the detection of syntactic clones can be done relatively easy, semantic clones are more difficult to detect [43]. This is especially true for test code, where semantics differ from production code. Production code usually performs some calculation which can be represented with a program dependency graph, while test code verifies the semantics of production code, verifying observable properties by means of assert statements.

In this paper, we will tackle the problem of semantic clone detection in test code by answering the following research questions:

— **RQ 1:** Is it feasible to detect semantic clones in test code? And, if yes, how does this compare to a syntactic clone detector?

— **RQ 2:** How do the clones detected by a semantic clone detector differ from those detected by a syntactic clone detector?

To answer these questions, we propose a novel method to detect semantically similar code clones in test code. We do this by using symbolic execution to generate a representation of test behaviour, which we then compare. We demonstrate our approach on the Apache Commons Math Library's test suite. The rest of this paper is structured as follows. In Section 5.2 we present the related work. In Section 5.3 we explain our approach and how we validated it. In Section 5.4 we present our results and in Section 5.6 we conclude.

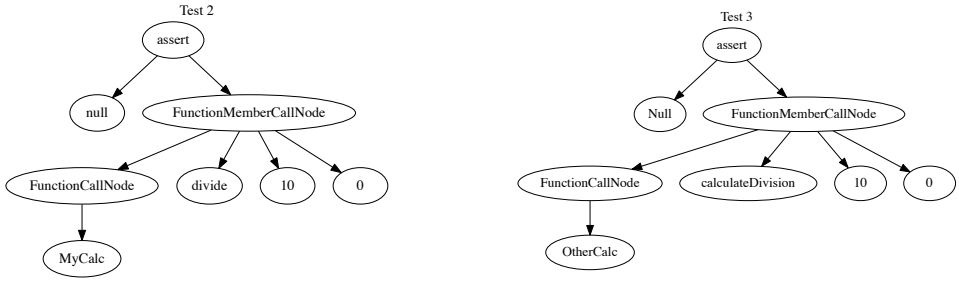

Figure 5.1: The test behaviour trees from the toy example.

5.2 RELATED WORK

There are many clone detectors that compare token strings, abstract syntax trees, or a combination of both. While these clone detectors are very useful for finding syntactic clones (type-I, type-II, or type-III), they are not capable of detecting semantic clones (type-IV) [43].

Komodoor et al. were the first to investigate a new approach to detect semantic clones. They proposed the use of program dependency graphs and program slicing [44]. Soon after, Krinke again showed how progam dependency graphs can be used to detect semantic clones [45]. This technique has since been implemented in a few tools, such as DECKARD and CCCD [46, 47].

Kim et al. proposed the use of static analysis to extract the memory states for each procedure exit point [48]. Another approach was proposed by Jiang et al. who apply random testing to detect similar function output [49]. Their techniques are the first to detect semantic clones without any syntactic similarities, yet they can not be applied on test code. For the moment it is not yet known whether it is feasible or worthwhile to detect semantic clones in test code [50].

5.3 EXPERIMENTAL SETUP

5.3.1 Detection Method

In order to detect semantic clones in test code, we compare the behaviour of the tests. Test code verifies observable properties of production code using assert statements. These assert statements define what is being verified by the test, and therefore define

Test 1:

```
MyCalc tester = new MyCalc();
int result = tester.divide(10, 1);
assertEquals(10, result);
```

Test 2:

```
MyCalc tester = new MyCalc();
int result = tester.divide(10, 0);
assertEquals(null, result);
```

Test 3:

```
OtherCalc testcalc = new OtherCalc();
assertNull(testcalc.calculateDivision(10, 0));
```

Figure 5.2: Example of how asserts define test behaviour.

the behaviour of the test. We demonstrate this with a toy example in Figure 5.2. The setup and stimulation in the first two tests are syntactically almost identical, yet something different is being tested: the first test verifies that a division produces the correct result, while the second test verifies that a zero division does not produce any result. Furthermore, the second and third tests are syntactically different, yet semantically the same. Therefore, the behaviour of the test is defined by what is verified (e.g. assert statements), and not by the setup or stimulation.

We use a *Test Behaviour Tree* (TBT) to represent the behaviour of a test. We defined this representation in our previous work on test refactoring [7]. This tree consists of the assert statements of a test and their backward slice. As an example, Figure 5.1 shows the test behaviour trees of the latter two tests from Figure 5.2. Note that, if a test contains multiple assert statements, that test will have multiple TBTs. Because we compare these trees, we detect semantic clones at an assert-level granularity.

5.3.2 Algorithm

For the construction of the test behaviour trees, we use our open-source tool T-CORE [8]. This tool was originally created to help developers refactor their test code by detecting non-semantic-preserving changes during refactoring. It constructs the TBT internally, and then uses these trees to detect changes in behaviour. In this study, we extracted the code that constructs the TBT and reused it for our clone detection purposes.

T-CORE uses symbolic execution to generate the test behaviour tree of each assert. This symbolic execution will pass over the code line by line, simulating the current memory state at every step. However, it utilizes a unique feature of unit test code, namely that most of the values of the variables can be deduced from the local code. This allows the symbolic execution to perform calculations for all operations performed on local variables, similar to how a compiler would perform constant folding [33]. Above that, the symbolic execution also evaluates loops, conditionals, and function calls. As a result, the symbolic execution can replace many of the symbols for variables with their actual value. When an assert statement is reached, it already has all the necessary data to create the backward slice of the assert statement, and the TBT can be constructed.

After we construct the test behaviour trees for each test, we perform a comparison of each tree with every other tree. This comparison requires the trees to be exactly the same, with exception of literals and identifiers. We allow the value of literals to be different, such that type-II clones can be detected. We also allow identifiers to be different, as long as the number of different identifiers does not exceed 5% of the total tree size. This threshold can be tweaked for a more strict or less strict comparison. However, we determined through the experiment that 5% is ideal, as increasing it generates many more false positives and decreasing it causes some obvious clones to be missed. We write the discovered clone pairs to an XML file.

5.3.3 Dataset

We evaluate our technique on the Apache Commons Math Library. This is an open-source Java library that implements solutions for many mathematical problems. Because it implements different algorithms for each problem, we know that it in fact contains type-IV code clones. For example, it implements 6 different algorithms that perform integration of a curve. While these algoritms are syntactically completely different, they are semantically identical. The library contains an extensive test suite, which has a size of 105 KLOC, consisting of 4475 unit tests, and has 90% line coverage. Since our approach only works on test code, this test suite will be our dataset.

Because T-CORE is currently still a prototype, it is unable to parse all files in the test suite. Some files contain advanced language features, such as the use of templated containers, while other files contain quadruple nested loops that timed out the symbolic execution. We excluded these files from our dataset. The final dataset consisted of 360 files from the Apache Commons Math Library's test suite, which contained 32 KLOC and 1613 unit tests.

5.3.4 Post-Processing

Because the symbolic execution analyses each iteration of each loop, some assert statements were added multiple times, and subsequently matched with themselves during our comparison. To solve this problem, we performed a post-processing step to filter out all unnecessary clone pairs. We implemented this filter in a python script, such that it can easily be replicated. After filtering out these duplicate clone pairs, we scale up our granularity to test size. If an assert from one test is matched with an assert from another test, it is likely the other asserts in those tests will be matched as well. This results in unique clone pairs of test granularity.

5.3.5 Evaluation

In order to answer our first research question, we do a quantitative evaluation of our approach. We investigate the number of type-IV clones detected. Above that, since type-I, type-II, and type-III clones are likely to have similar semantics as well, our approach should also detect them. We investigate the number of these clones detected as well.

To provide more insight for this quantitative evaluation, we compare our approach with a state-of-the-art syntactic clone detector. We use the NiCad clone detector [51]. NiCad is a scalable and flexible state-of-the-art clone detection tool which implements a hybrid clone detection method. It takes as input a source directory and provides output results in XML. We made sure that our XML format matched the output generated by NiCad, to allow for easy analysis.

Once we have all clones from both T-CORE and NiCad, we classify them according to type: type-I to type-IV, or as false positive. This classification is performed manually by the authors by inspecting both tests of each clone pair. We used the definitions of the types specified by Roy et al. to solve dubiety [43]. In order to help this process, we use a python script that automatically opens the source of both tests side-by-side and highlights all syntactic similarities.

In order to answer our second research question, we split up the clones found by T-CORE and NiCad into three categories: (a) clones that were found only by T-CORE, (b) clones that were found by both tools, and (c) clones that were found only by NiCad. This, combined with the type-classification, allows us to perform a qualitative evaluation of the clones found. This evaluation provides insight in how clones detected by a syntactic clone detector differ from those detected by a semantic clone detector.

Table 5.1: Clones found classified by type.

Tool	**T-I**	**T-II**	**T-III**	**T-IV**	**FP**	**Total**
T-CORE	52	385	44	259	15	755
NiCad	13	50	27	100	0	190
Category						
a) T-CORE only	40	363	36	205	15	659
b) T-CORE & NiCad	12	22	8	54	0	96
c) NiCad only	1	28	19	46	0	94

5.4 RESULTS

RQ1: *Is it feasible to detect semantic clones in test code? And, if yes, how does this compare to a syntactic clone detector?*

Table 5.1 provides an overview of the number of clones detected by each tool categorized by type. In total, we found 755 clone pairs using T-CORE. Of these 755 clones, our classification found that only 15 are false positives (column FP). Therefore we state that our approach has a precision of 98%. NiCad found 190 clone pairs, which is significantly less than our approach. However, NiCad did not report any false positives, and thus has a higher precision. We can conclude that it is indeed feasible to detect semantic clones with a relatively high precision using our approach.

The 15 false positives are generated for two reasons. First, the test suite contains some asserts that compare two local variables. Since the symbolic execution calculates the value of these variables, and since we do not check these values to allow the detection of type-II clones, such asserts are too general to detect clones correctly. This could be solved by introducing a minimum tree size parameter in the detection algorithm, similar to a minimum number of lines or tokens used in other clone detectors. Secondly, some asserts are used at the start of the test to verify that the set-up was correct, before the *Unit Under Test* (UUT) is exercised. While these asserts are clones when considering them at an assert-level granularity, they become a false positive when comparing the tests at function-level granularity.

RQ2: *How do the clones detected by a semantic clone detector differ from those detected by a syntactic clone detector?*

Of the 755 clones detected by T-CORE, we notice that 385 are of type-II. These clones are common in test code, as one function from production code will be tested by multiple unit tests which differ only in input values. Figure 5.3 shows an example of such a typical type-II clone in test code. We note that NiCad detects significantly fewer clones

```
final double[] a = { 1, 2, 3, 4 };
final double[] b = { -5, -6, 7, 8 };
final double expected = 8;
assertEquals(expected, distance.compute(a, b));
assertEquals(expected, distance.compute(b, a));
```

```
final double[] a = { 1, -2, 3, 4 };
final double[] b = { -5, -6, 7, 8 };
final double expected = 18;
assertEquals(expected, distance.compute(a, b));
assertEquals(expected, distance.compute(b, a));
```

Figure 5.3: Example of a typical type-II clone in test code.
(differences marked in bold and underlined)

of this type compared to T-CORE, even though it should be able to detect the syntactical similarities of a type-II clone easily. This anomaly is caused by the size of the cloned unit tests. Unit tests are often relatively small. We ran NiCad using the default settings, which include a minimum clone size of 10 lines, but a lot of unit tests are much smaller than that. Upon inspection of the type-II clones found by T-CORE, we noticed that a significant number of these clones are smaller than 5 lines. Running NiCad with a minimum size of 5 lines or less would generate too many false positives. In fact, any syntactic detector would generated many false positives with such small sizes. This shows that it is worthwhile to detect semantic clones in test code.

When we look at the total number of type-III clones found by each tools, the difference is relatively small, with T-CORE detecting 44 and NiCad detecting 27 type-III clones. However, we see that only 8 of these clones are common between both tools. This indicates that there are differences in the type-III clones found by each tool. We first inspect the clones that only NiCad detects and find that these cloned tests setup and stimulate the UUT in a similar way, but then verify a different aspect of the UUT. Therefore, we can conclude that the semantics of the tests are different, which is why T-CORE does not detect them as clones. When we inspect the clones that only T-CORE detects, we find two interesting cases. First, we discover a test containing inline function definitions. These functions are used as a mock for the UUT. Because they are defined inline, they create a gap between the setup of the UUT and its stimulation. While type-III clones are allowed to have gaps by definition, the inline functions create a too large gap for NiCad to detect. Secondly, we discover a few clones where the setup and oracle part of the tests are similar, but the stimulation of the UUT is different. More specifically, these tests verify that special cases would not cause the UUT to become inconsistent. This again causes a gap between the syntactically similar parts that is too large for NiCad to detect.

```
TestProblem1 pb = new TestProblem1();
FirstOrderIntegrator integ = new HighamHall54Integrator(minStep, maxStep,
    scalAbsoluteTolerance, scalRelativeTolerance);
TestProblemHandler handler = new TestProblemHandler(pb, integ);
integ.addStepHandler(handler);
integ.integrate(pb,
  pb.getInitialTime(), pb.getInitialState(),
  pb.getFinalTime(), new double[pb.getDimension()]);
Assert.assertTrue(handler.getMaximalValueError() < (1.3 * scalAbsoluteTolerance));
Assert.assertEquals(0, handler.getMaximalTimeError(), 1.0e-12);
```

```
TestProblem1 pb = new TestProblem1();
EmbeddedRungeKuttaIntegrator integ = new DormandPrince54Integrator(minStep, maxStep,
    scalAbsoluteTolerance, scalRelativeTolerance);
TestProblemHandler handler = new TestProblemHandler(pb, integ);
integ.setSafety(0.8);
integ.setMaxGrowth(5.0);
integ.setMinReduction(0.3);
integ.addStepHandler(handler);
integ.integrate(pb,
  pb.getInitialTime(), pb.getInitialState(),
  pb.getFinalTime(), new double[pb.getDimension()]);
Assert.assertTrue(handler.getMaximalValueError() < (0.7 * scalAbsoluteTolerance));
Assert.assertEquals(0, handler.getMaximalTimeError(), 1.0e-12);
```

Figure 5.4: Example of a type-IV clone.
(differences marked in bold and underlined)

Of the 755 clone pairs found by T-CORE, 259 are of type-IV clones. Figure 5.4 shows an example of such a type-IV clone. In this example, two different integrators are being tested. The integrators themselves, as they are implemented in production code, are type-IV clones: they each implement a different algorithm for integration of a curve. These two tests verify the same properties of the different integration algorithms on the same curve. We state that the semantics of the test (what is being verified) is the same, yet how it is being done (algorithm and its parameters) is different. Therefore, these two tests are also type-IV clones.

Even though there are some syntactic similarities, mostly in the setup and assert statements, NiCad can not detect these due to the large gap created by the stimulation of the UUT. NiCad does detect a relatively large number of type-IV clones. However, these clones are similar to the one in Figure 5.4, but with the gap between the syntactically similar parts of the test small enough for NiCad to detect it as a type-III clone. We classify these clones as type-IV clones, since we consider them at a test granularity. These type-IV clones indeed contain a smaller type-III clone, similar to how type-III clones contain smaller type-II clones, and type-II clones contain smaller type-I clones, yet we always classify them as the largest possible type given the granularity level.

5.5 THREATS TO VALIDITY

5.5.1 Internal Validity

The manual classification of the discovered clones is a threat to internal validity. In order to minimize this threat, we strictly applied the definitions of the types specified by Roy et al. [43]. We also used a python script that automatically opens the source of both tests side-by-side and highlights all syntactic similarities to minimize the mental load. Yet, we cannot guarantee that no mistakes were made during this classification.

5.5.2 External Validity

In this work, we performed an initial investigation towards the feasibility of our approach. Because this initial investigation was limited in size, a threat exists that our conclusions are not generalizable. In future work, we will perform the same experiment on a larger dataset and compare to other clone detection tools.

5.6 CONCLUSIONS

In this paper we proposed a novel method to detect semantic code clones in test code. We do this by using symbolic execution to generate a representation of test behaviour, which we compare. We demonstrated on the Apache Commons Math Library's test suite that our approach detects 755 clone pairs with a precision of 98%. We also showed that 259 of the 755 detected clone pairs are type-IV clones. This confirms that it is both feasible and worthwhile to investigate semantic clones in test code.

Part II

Comparing Code Clones in Test Code and Production Code

Chapter 6

Comparitative Study of Test Code Clones and Production Code Clones

A Comparative Study of Test Code Clones and Production Code Clones

Brent van Bladel and Serge Demeyer
In *Journal of Systems and Software (JSS)*, 110940. June, 2021.

This chapter was originally published in the *Journal of Systems and Software (JSS)* as an extension of our previously published paper **Clone Detection in Test Code: an Empirical Evaluation** in *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

CONTEXT
In this chapter, we tackle our second main research question by directly comparing duplication in the production and test code of open-source projects. We focus our analysis on the latest version of each project, and as such only provide findings concerning the latest snapshot of the projects. Yet, we are already able to provide many interesting insights in the differences between test clones and production clones.

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.

6.1 INTRODUCTION

The recent popularity of agile software development has increased the emphasis on software testing for developers. In particular, test-driven development [52] and continuous integration [53, 54] require an effective test suite, which is executed early and often [55]. 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 [3, 4].

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 [56, 57]. 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 [5]. 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 [41]) can still cause clones accidentally. The prevalence of clones in test code has been shown by Hasanain et al. [58] and was confirmed in our prior work [11].

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 [22]. 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 [11] 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 a small number 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 6.2 provides the required theoretical background while Section 6.3 lists the related work. Section 6.4 describes the experimental set-up, which naturally leads to Section 6.5 reporting the results. Section 6.6 lists avenues for further work, both from a research and tool builders perspective. Section 6.7 enumerates the threats to validity, and Section 6.8 concludes the paper.

6.2 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.

6.3 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 [6]. 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 [59]. 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) [60]. 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 [6, 59, 61].

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 [6, 59, 61]. When analysing the prevalence of clones this is usually done by comparing the *clone density* (also known as clone percentage [12], or TCMp or TCLOCp depending on the granularity [13, 14]). When researchers compare clone detection tools, they calculate the precision and recall [59]. 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* [62].

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 [22]. 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 [57]. 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 [63].

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 [64]. 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 [58]. 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 [11].

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.

6.4 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.

6.4.1 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 [59].

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 [44, 45]. Other techniques include static analysis of memory states at each procedure exit point [48], or applying random testing to detect similar function output [49]. More recently, machine learning techniques, such as deep neutral networks, have been successfully used to detect more difficult Type III and Type IV clones [65, 66].

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* [12] 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 6.4.3). For example, *Deckard* [67] 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* [68], SourcererCC [69], and CloneWorks [70] were considered, but their output was not easily converted to our reference format (see Section 6.4.4).
- *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* [9] 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 [51, 71].
- *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 [59].
- *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 [72, 73].

6.4.2 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 6.1 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 6.1: 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.

6.4.3 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

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

Figure 6.1: Example of a clone pair in the reference XML format.

test code, as this allows for the smaller duplicated tests to be detected without generating many false positives [9]. 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.

6.4.4 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 6.1 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 6.4.5).

6.4.5 Classification

After postprocessing, we performed type classification on all detected clones. Due to the large number 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.

6.4.6 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 [58]. Our previous work confirmed the prevalence of clones in test code by analyzing three open-source systems [11]. 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 [12], or TCMp or TCLOCp depending on the granularity [13, 14]) 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 [5]. 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 [41]) 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 number 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) [74, 75]. 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 [76]. 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 [41]) 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 [60, 77]. However, these benchmarks focus only on production code and do not contain test code [61]. 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 [62]. The relative recall also provides us with the percentage of false negatives, since it can be calculated as

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

Table 6.2: 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	-

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.

6.5 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 6.2 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 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 number of clones in the production code, with the clone density in test code still more than twice of that in production code.

Table 6.3: 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%

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 6.3 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.

Test code has an increased number 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 6.4 provides an overview of the clone classes detected by the three tools in each dataset. The column labelled *Classes* shows the number 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 number of such fragments is consistently higher for test code compared to production code, most

Table 6.4: 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%

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 number 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 number of clone classes in test code compared to production code.

Figure 6.2 shows the boxplot of the size of the clone classes for each datasets. Or, in other words, it shows the number 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 6.3 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.

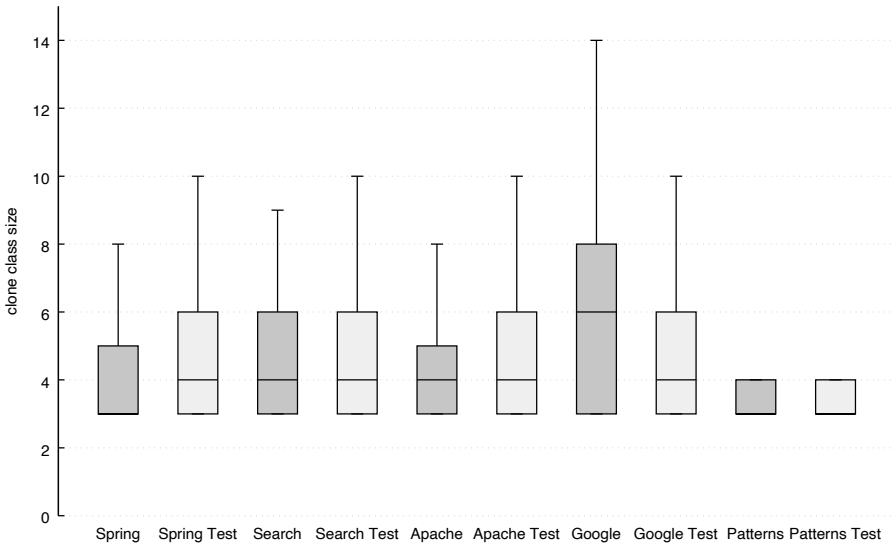

Figure 6.2: Clone class sizes for each dataset.

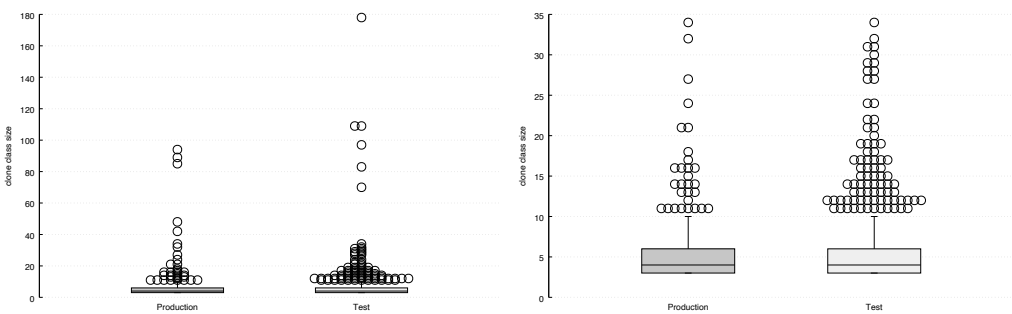

Figure 6.3: Clone class sizes for production code and test code, with outliers.
(The left figure shows the full overview, while the right figure zooms in on the central clusters.)

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?*

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 6.4 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 6.4 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 6.4: Example of a typical Type II clone in test code, from the Search test dataset.
(differences marked in bold and underlined)

For the same reason, the typical structure of unit tests also cause many Type III clones. Figure 6.5 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 6.5: Example of a typical Type III clone in test code, from the Spring test dataset.
(differences marked in bold and underlined)

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 6.6 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 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 Pat-

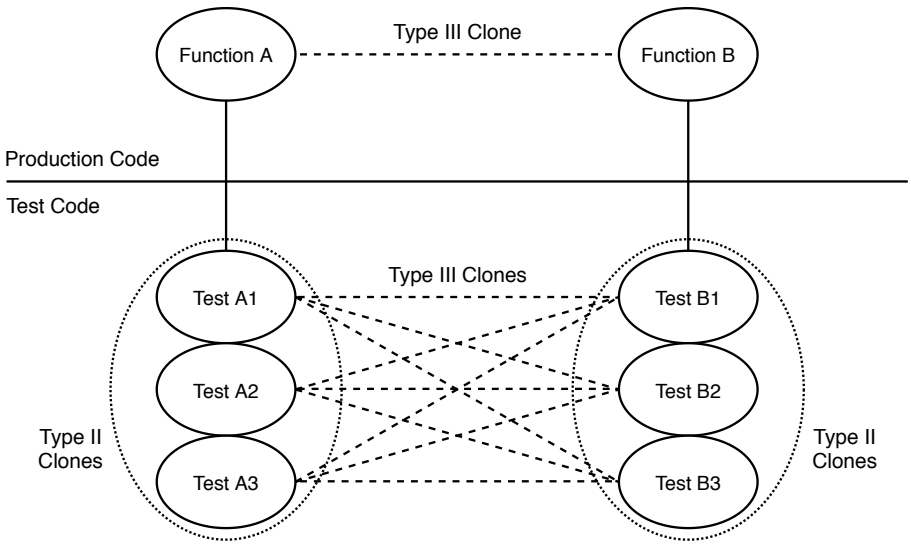

Figure 6.6: Relation between test clones and production clones.

terns 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 6.5 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.

When we look at relative recall, we can see that NiCad's text-based technique detects the most syntactic 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,

Table 6.5: Overview of the precision and relative recall for each clone detector.

Project		Precision			Relative Recall		
		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%

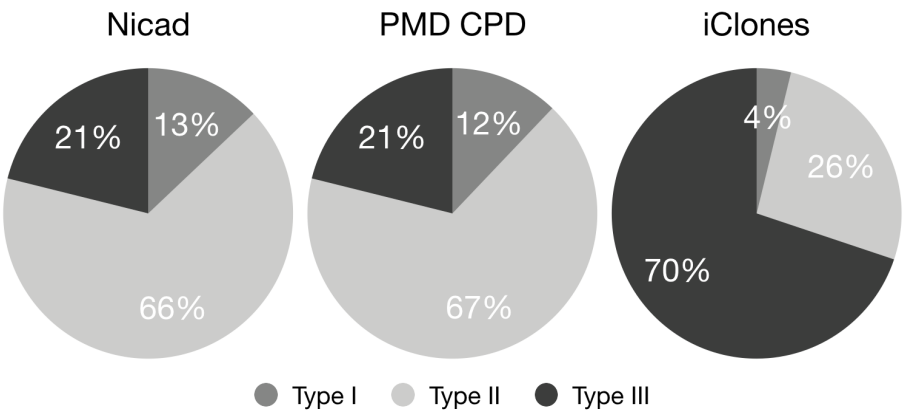

Figure 6.7: Clones of each type detected by the different tools.

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 6.7, which provides an overview of the total number 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.

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 number 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 number of Type III clones.
⇒ From a tools perspective, specific fine-tuning for test code is needed.

6.6 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 [78]. 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 [79]. 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 [80].

Test Amplification Test amplification is the act of automatically transforming a manually written unit-test to exercise boundary conditions [81]. 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 [82] which has been replicated for Pharo/Smalltalk within our lab under the name of SmallAmp [83]. 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 [77]. 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 [60, 84]. 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 [85]. Rather than transplanting tests for clones in the system under test (like advocated by Zhang et al. [86]), 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 [11] 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 number 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 par-

ticular 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 false positives anyway) to exploit the consistent structure of unit test code (the *setup-stimulate-verify-teardown* (S-S-V-T) cycle [41]).

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 [87]. Several such consistent change recommenders have been created over the years, we list just a few: CloneTracker [88], CodeCloningEvents [89], Clone Change Notification [90], Clone Notifier [91]. 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. It could, for example, allow developers to consistently change every test for a single unit, whenever this unit under test changes its interface or behaviour.

6.7 THREATS TO VALIDITY

6.7.1 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.

6.7.2 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.

6.8 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 number 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.

Chapter 7

Comparitative Study of Clone Evolution in Test Code and Production Code

A Comparitative Study of Clone Evolution in Test Code and Production Code

Brent van Bladel and Serge Demeyer
In *Proceedings of the 6th Workshop on Validation, Analysis and Evolution of Software Tests* (*VST*). Macao, China. March, 2023.

This chapter will be published to *Proceedings of the 6th Workshop on Validation, Analysis and Evolution of Software Tests* (*VST*).

CONTEXT
In this chapter, we tackle our second main research question by comparing the full evolution of duplication in production code and test code. By focussing our analysis on the entire evolution of the code clones, we get a full picture of the differences between test clones and production clones, and are confident that we can provide a thorough answer to our second main research question.

ABSTRACT *Recent research has shown that clones occur far more often in test code than in production code. This is explained by the typical template of unit test code (the setup-stimulate-verify-teardown cycle), a template which lends itself well to parameterisation of proven solutions. However, little is known on how these clones evolve over time. In this paper, we compare the evolution of clones in test code with those in production code. Based on a quantitative and quali-*

tative study of eight representative open-source systems, we conclude that clones in test code are inherently different than clones in production code throughout development. As a consequence, consistent maintenance and tracking of test clones becomes more important, hence the need for special purpose clone tracking and management tools.

7.1 INTRODUCTION

Software testing provides important information on the efficacy as well as efficiency of software applications. The test suite is often used as a final quality gate, and as such it plays a crucial role in the success of an application [2]. With agile teams working incrementally on the production code, the test code needs to be updated, extended, and maintained each iteration. Therefore, it is a recommended practice to continuously monitor the quality of the test suite [3, 4].

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 may cause duplicate tests (a.k.a. test clones), as the quickest way for a developer to test a new feature is to copy, paste, and modify an existing test [5]. 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 [41]) induces a template which easily causes variations on a given theme. These variations then surface as clones with small differences in literal values to exercise boundary values (a.k.a. copy-paste clones), or extra statements to bring the component under test in a given state (a.k.a near-miss clones). Previous research has already reported the prevalence of clones in test code [11, 58]. In follow-up research, we showed that test code contains more than twice the amount of duplication compared to production code, as duplicated functionality in production code causes a significant increase in test clones [10]. Moreover, we found that test clones inherently differ from clones in production code, with unit tests often being duplicated many times with small variations, confirming that this duplication is caused by the S-S-V-T cycle [10].

In this paper, we further investigate this inherent difference between clones in production and test code by analysing how they evolve throughout software development. The idea of studying clone evolution originated in 2005, when Kim et al. coined the term “Code Clone Genealogies" for describing how a family of clones evolves over time [92], which has led to many interesting insights. With this study, we extend these insights to test code.

Based on a quantitative and qualitative comparison of the evolution of clones in eight representative open-source systems, we make the following observations.

1. The amount of duplication in test code is significantly higher than in production at every point during development.
2. Clone density in test code is much more sensitive to changes in the code base, while clone density in production code is far more stable.
3. Overall the survivability of test clones is similar to that of production clones. However, when clones do not survive, they disappear sooner in test code than in production code.
4. Clones in test code are being changed slightly more often than in production code, yet they are more likely to be changed consistently than inconsistently.

These four observations confirm that test code clones do indeed inherently differ from production clones, not only at their instantiation, but at every point of their evolution. It implies that resource allocation and time estimation for clone-related maintenance activities must take into account the special nature of test code. Moreover, consistent maintenance and tracking of test clones becomes more important, hence the need for special purpose clone tracking and management tools increases.

The remainder of this paper is organised as follows. Section 7.2 provides the required theoretical background while Section 7.3 lists the related work. Section 7.4 describes the experimental set-up, which naturally leads to Section 7.5 reporting the results. Section 7.6 enumerates the threats to validity, and Section 7.7 concludes the paper.

7.2 TERMINOLOGY

In this section we will provide definitions of the used terminolgy in this paper. These definitions were adopted from Kim et al. [92].

Clone fragment. One fragment of code that is duplicated is called a clone fragment.

Code clone. When two or more 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 are used interchangeably in this paper.

Change pattern. A change pattern describes how clone fragments change from one version of the system to the next. We differentiate three different change patterns in this paper.

— *Same:* The code of every clone fragment in the clone remains the same. Development activities (e.g. bug fix, refactoring) did not alter the duplicated code.

— *Consistent:* A certain change was made to every clone fragment in the clone. Some development activity altered the duplicated code, and as such had to be applied to every fragment of the clone.

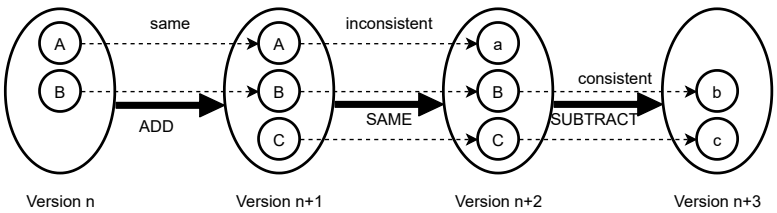

Figure 7.1: Example of a clone lineage. The bold arrows show the evolution pattern, the dotted arrows show the change pattern.

— *Inconsistent:* A certain change was made to some clone fragments in the clone. Some development activity altered the duplicated code, but was not applied to every fragment of the clone.

Evolution pattern. An evolution pattern describes how the number of clone fragments changes from one version of the system to the next. We differentiate three different evolution patterns in this paper.

— *Same:* The number of code fragments in the clone remains the same.

— *Addition:* The number of code fragments in the clone increases. Some development activity added or changed code outside the clone such that the code matches the duplicated code.

— *Subtraction:* The number of code fragments in the clone decreases. Some development activity removed or changed code inside the clone such that the code no longer matches the duplicated code.

Clone lineage. A clone lineage is the ordered set of all versions of one clone throughout development. In a clone lineage, a clone in version n of the system is connected with a clone in version $n - 1$ by a change and evolution pattern. Figure 7.1 shows an example of a clone lineage, as well as an example of every change and evolution pattern.

Clone genealogy. A clone genealogy is a set of clone lineages that have originated from the same clone. Figure 7.2 shows an example of a clone genealogy of two lineages with a common origin. Note how the change and evolution pattern can differ between lineages when they split from the common line.

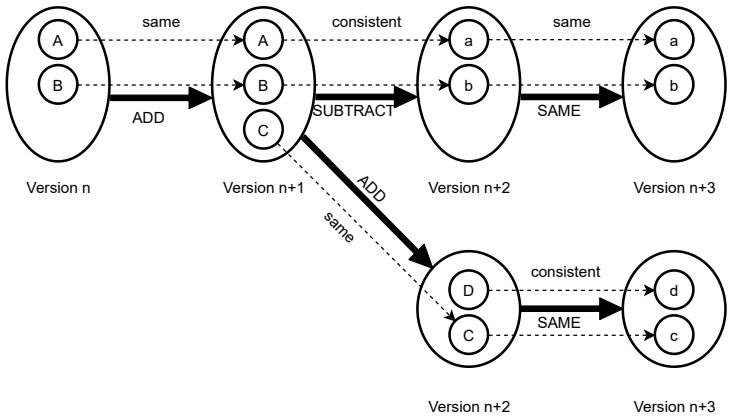

Figure 7.2: Example of a clone genealogy. The bold arrows show the evolution pattern, the dotted arrows show the change pattern.

7.3 RELATED WORK

A lot of research has already been performed on software clones. In 2007, Koschke performed a large survey of the literature on software clones, summarizing the state of clone detection research [6]. 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 [59]. Over the years, many studies have 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 [6, 59, 61].

In our previous research, we were the first to consider clones in test code separately from clones in production code in a comparative study of the two [10]. We found that code clones in test code inherently differ from those in production code, mainly caused by the specific structure of unit tests. This motivated us to investigate this difference further, by analysing the evolution of test clones from the perspective of clone genealogies.

The idea of studying the evolution of code clones was first introduced in 2005 by Kim et al. [92]. They defined the terms clone genealogy and clone lineage, and proposed a set of 6 patterns that describe their evolution. They also implemented a tool to extract

this evolution data [93]. In an initial study, they applied their tool on two projects and concluded that clones impose obstacles during software evolution and that popular refactoring techniques cannot easily remove many long-lived clones.

In 2007, Jens Krinke investigated changes made to code clones in evolving software [94, 95]. He found that usually half of the changes to code clones are inconsistent changes, and when there are inconsistent changes to a code clone in a version, it is rarely the case that there are additional changes in later versions to make the clone group consistent again. A year later, he found that generally cloned code is changed less often than non-cloned code [96], indicating that clones indeed impose obstacles during software evolution.

By 2009, several different techniques for tracking code clones throughout software development had been proposed [97, 98, 99]. This opened the door to many more studies on clone evolution, all of which confirmed that clones are rarely changed and, when they are changed, the change is most likely inconsistent [100, 101, 102, 103]. Other studies started to focus on how harmful long-living clones and inconsistent changes to clones actually are [104, 105, 106, 107]. They found that clones may not be as detrimental in software maintenance as previously thought, as the inconsistent changes are made intentionally to adapt the clone to its environment. They propose that instead of aggressively refactoring clones, the focus should be on tracking and managing clones during the evolution of software systems.

As the focus shifted towards tracking and managing clones, more tools and techniques were proposed for this purpose [108, 109, 110, 111]. The research on clone evolution also shifted towards more practical applications, such as investigating late propagation of clones [112, 113, 114], studying fault-proness and bug propagation of clones [115, 116, 117], and even predicting changes to clones [118]. Yet no research has been performed on the evolution of clones in test code.

7.4 EXPERIMENTAL SETUP

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

7.4.1 Clone Detection Tools

We selected *NiCad* and *iClones* as our clone detection tools. Both are fast tools that are easily configured for our purposes [59]. NiCad detects code clones using a text-based approach, while iClones uses a tree-based approach. By combining the results of two tools with two different approaches, we are able to detect a larger and more divers number of clones.

We opt for the default configuration of the tools, as we assume that the default configuration would be best suited for a general purpose. There is only one parameter which we change: granularity. We set the granularity to function-level, meaning that each clone fragment will consist of a function containing the cloned code. Since every unit-test is implemented as a function, clone detection on the test suite will happen on a test-level granularity.

7.4.2 Dataset

For our study, we selected four open-source Java projects and four open-source C projects from GitHub. For Java, we chose the Google Guava library, the Apache Commons Math library, the Apache Avro library, and the Eclipse vert.x toolkit. For C, we chose the Cairo graphics library, the Apache APR project, the libarchive library, and the libsodium library. These projects were selected because they are popular and commonly used open-source projects with extensive test suites. All eight projects make use of a continuous integration (CI) server that runs the test suite after each commit. At the time of writing,[1] all projects pass their CI build.

We use the production code and the test suite of these projects as the dataset for our study. Table 7.1 and Table 7.2 shows an overview of the size of each project at the time of writing. The size is presented in terms of functions (for the production code), tests (for the test code), and lines of code (for both). Note that the lines of code (LOC) metric does not include comments or blank lines. The number of commits and the number of analysed versions are also included in the tables.

7.4.3 Data Collection

In order to construct code clone genealogies, we need cloning data from the whole software evolution history. We implemented a Python script to collect this data.[2]

[1] March 2022

[2] All scripts and data used are available in our replication package: DOI 10.6084/m9.figshare.17206997.v3.

Table 7.1: Descriptive Statistics of Java projects.

Name		Functions or Tests	LOC	Total Commits	Analysed Versions
Google Guava	Prod	4,773	91,287	5,785	577
	Test	13,822	159,776		
Apache Math	Prod	4,741	70,640	6,479	647
	Test	5,168	75,567		
Apache Avro	Prod	2,524	35,578	2,734	272
	Test	2,769	40,010		
Eclipse vert.x	Prod	2,737	59,702	3,053	304
	Test	784	72,873		

Table 7.2: Descriptive Statistics of C projects.

Name		Functions or Tests	LOC	Total Commits	Analysed Versions
Cairo	Prod	5,241	160,037	10,842	1,084
	Test	1,488	41,048		
Apache APR	Prod	1,247	56,505	9,070	907
	Test	672	21,921		
libarchive	Prod	2,102	96,792	4,620	461
	Test	731	52,701		
libsodium	Prod	871	19,208	3,536	353
	Test	136	16,679		

A. Analysed versions

First, the Python script gathers the commit history of the project via Git. This results in a list of the short Git version hash for each commit, which can be used to switch between different versions of the project.

Because clone detection is a time-consuming process, it would not be feasible to perform it on every version of the project. Moreover, some commits do not alter the code (e.g. adding documentation or changing configuration files), so two consecutive versions could be programmatically the same, which would not be interesting for our analysis. Therefore, we decided to only analyse project versions according to a constant interval of 10 commits. We found this interval large enough to make clone detection feasible while still small enough to capture changes in the cloning data. The final number of analysed versions for each project can be found in Table 7.1 and Table 7.2.

B. Clone detection

For every version we want to analyse, the Python script first divides the project into two parts: the production code and the test code. We make this division based on the naming convention of the source files: each file that has the word "Test" in its name or path is part of the test code, every other file is part of the production code. We manually verified that the projects in our dataset all followed this naming convention.

The Python script then runs the NiCad and iClones clone detector tools on the production code and the test code separately. This means that all detected clones are completely contained within either the production code or test code of the project. We merge the detected clones from both tools, in order to have a larger dataset of clones for our analysis. Each clone consists of multiple clone fragments, for which the filename and line numbers are extracted.

The Python script then extends this data with additional information needed for post-processing. For every clone fragment, it adds the name of the function containing the clone fragment. Note that each clone fragment is contained in exactly one function, since we detect clones on a function-level granularity. The fragment's code is then normalized (i.e. removing whitespace and comments) and hashed. The resulting hash is also added to every clone fragment in the XML file.

C. Post-processing

After collecting the clone data at every version we want to analyse, the Python script performs several post-processing steps to construct the clone genealogies. The post-processing steps that construct clone genealogies are based on the original methods defined by Kim et al. and Bakota et al. [92, 97].

First, we need to track the clones that appear throughout multiple versions of the project. This is done by chronologically traversing the analysed versions, comparing the filename and function name of clone fragments with those of the next analysed version. We assume that every function in a certain file has a unique name, and therefore that, if a function *foo* in a file *bar* is a clone fragment in two consecutive analysed versions, they are the same clone fragment. By tracking the clone fragments throughout the versions, we can easily match the clones containing those fragments. When we have all instances of a clone throughout the different analysed versions chronologically, we call it a clone lineage. Finally, we can group clone lineages that have a common origin to construct the clone genealogies.

Once we have the clone genealogies, we annotate them with the evolution pattern and change pattern. To determine the evolution pattern, we count the number of clone fragments contained in two consecutive versions. If this number increases or decreases, there was an *addition* or *subtraction* respectively. To determine the change pattern, we compare the hashes of the fragments' normalized code from two consecutive versions. If some of the hashes have changed, but not all, there was an *inconsistent* change. If all of the hashes have changed, there was a *consistent* change. We perform this annotation for every version in every genealogy.

7.4.4 Research Questions

In this paper we analyse how code clone genealogies in production code differ from those in test code. To guide us through our analysis, we propose three research questions. In this section, we motivate why we investigate these research questions and explain the approach we use to answer them.

RQ1: *How do the clone density for production code and test code evolve throughout development?*

Motivation: In our previous research, we have shown that test code contains more than twice as much duplication as production code [10]. However, we only considered the latest version of each project. It is unknown how the duplicated code evolved to get to that point, whether the amount of duplication in test code was higher than production throughout development or if it only peaked near the end. Investigating how the clone density evolves through time will not only provide more insight in the results of our previous research, but also in developers coding, testing, and cloning practices.

Approach: To answer this research question, we calculate the clone density for both production and test code at every analysed version. Clone density (also known as clone percentage [12], or TCMp or TCLOCp depending on the granularity [13, 14]) 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. Since we detect clones on a function-level granularity, f_c is equal to the number of unique clone fragments.

By constructing a timeplot of the clone density for both production and test code, we can compare the evolution throughout development. To complement this data, we also inspect the code whenever a sudden change in clone density was recorded, in order to find the cause of these anomalies.

We also determined the clone density in terms of LOC, defined as

$$clone\ density = \frac{LOC_c * 100}{LOC_{tot}}$$

where LOC_c denotes the number of cloned lines of code, and LOC_{tot} refers to the total number of lines of code in the system. However, since there is a direct correlation between the density in terms of LOC and the density in terms of functions, we omit the former from our results in favor of a closer analysis of the latter. Yet, our conclusions hold for both. The omitted density data is available in our replication package.

RQ2: *Do clones in test code live longer compared to clones in production code?*

Motivation: Even though analysing the total amount of duplication at each version can provide useful insights (RQ1), it does not tell us anything about the actual clones. Therefore, in RQ2 and RQ3, we focus our investigation towards the evolution of clone geneologies throughout development.

First we look at how long clones survive in the system, again making the comparison between production and test code. We know that test code contains more than twice as much duplication as production code, but it is unknown if clones in test code also survive longer, or if this additional duplication is caused by many shortlived clones. Answering this research question will help developers better understand how to maintain clones, since depending on the answer, a different maintenance strategy is required.

Approach: To answer this research question, we first divide the clone geneologies into two categories, *alive* and *dead*. Alive genealogies are still in the system in the final version, while dead genealogies do not appear in the final version [92]. Since alive genealogies are still currently evolving, they cannot be used to determine how long the clone will survive. Therefore, we exclude clones that are alive in our analysis, focussing only on the dead genealogies. More specifically, we gather data on *k-volatile* genealogies. A k-volatile genealogy is a dead genealogy that has an age lower or equal to k [92]. Using a cumulative distribution function to sum all k-volatile genealogies for every k, we can calculate the relative number of clones that survive up to a certain age. By doing this for both production and test genealogies, we can compare the age of test clones with that of production clones.

RQ3: *How do clones in production code and test code evolve according to the evolution and change patterns?*

Motivation: After investigating how long clones live, we focus our research to how clones evolve while they live. More specifically, how clones evolve according to the evolution and change patterns. Our previous research showed that test clones are inherently different from production clones [10]. However, it is unknown if they also evolve differently, if they change consistently or inconsistently more or less often, or if they grow or shrink more or less often, compared to production clones. Depending on the answer of this research question, refactoring and maintenance efforts might need to be prioritized for either production or test clones.

Approach: To answer this research question, we use the evolution pattern and change pattern annotations added during post-processing. We first use the change pattern annotations to calculate the relative ratio of lineages that never changed, lineages that only changed consistently, and lineages that changed inconstistently at least once. This will provide an indication on how stable the clones are. Then, we calculate the relative ratio of the evolution and change pattern annotations, such that we can get an idea of how the clones evolve. If we do this for both the production code and test code of each project, we can compare this evolution.

7.5 RESULTS AND DISCUSSION

In this section, we present the results from our study and formulate answers to our research questions.

7.5.1 RQ1: *How Do the Clone Density For Production Code and Test Code Evolve Throughout Development?*

Figure 7.3 shows the evolution of clone density for the eight projects under study. We can see that, generally, the amount of duplication in test code is significantly higher than that in production code at every point during development, except during the very first few commits of the projects when testing has not started yet.

The only exception is the Apache APR project. This project added their first tests around commit 2500, which causes the amount of test duplication to rise, reaching a maximum of 17% around commit 4000. However, the clone density of the test code then suddenly drops to 10.5%. After investigating the Git history, we found that this sudden

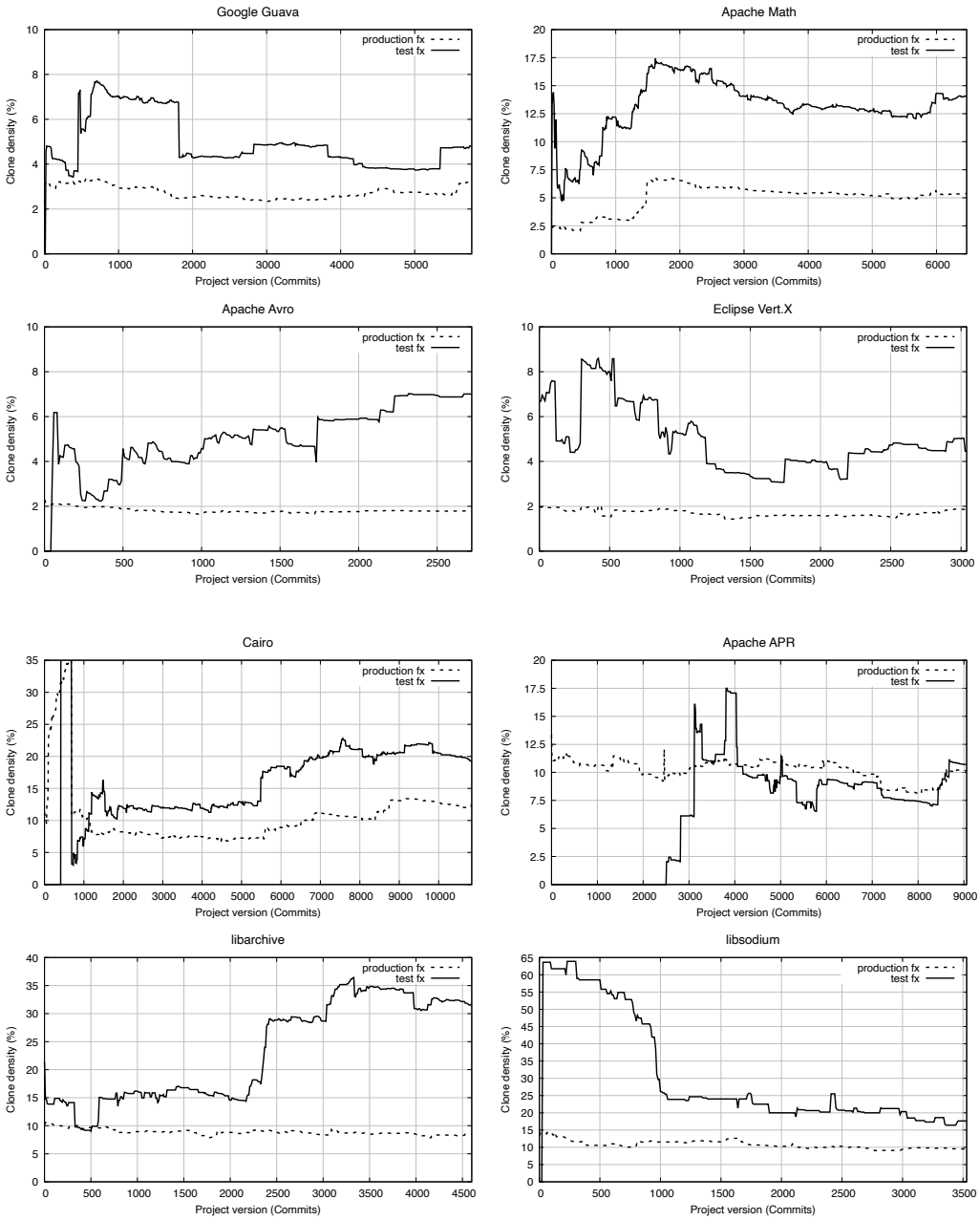

Figure 7.3: Evolution of clone density throughout project development. The upper four are Java projects, the lower four are C projects.

drop is caused by the deprecation of one of their API's, which lead to the removal of many duplicated API tests.[3] Throughout the rest of the project, the test code clone density remains around 10%, following a similar trend to the production code clone density.

Other projects also show the clone density of test code following the same trend as the clone density of production code, for example in the Apache Math and Cairo projects. This could be a result from the observation from our previous research, where we noted that clones in production code induce multiple clones in test code [10]. For example, if two functions *foo* and *bar* in production code are clones, multiple unit tests in *fooTests* and *barTests* will likely also be clones. Thus, a change in the production code clone density will result in a similar change in the test code clone density. We do observe that these changes in production code clone density only occur occasionally, as in most projects the production code clone density is very stable. It generally evolves very slowly over time, or often even remains around the same level throughout the entire project.

Test code clone density on the other hand is less stable. There are quite a few occurrences where it undergoes a significant change in a limited amount of time. We already discussed this for the Apache APR project. Another notable example is around commit 1800 of the Google Guave dataset, where the clone density for test code suddenly drops from 6.7% to 4.2%. When looking at the Git history, we found that one commit added a large set of hidden internal tests to the public project repository.[4] This commit added 73,306 LOC spread over 252 files, increasing the total number of tests from 11,914 to 18,994. However, many of these new tests were very small (2 - 4 LOC), and since our detection tools were configured with the default minimum clone size of 10 LOC, these tests were not considered during clone detection. As such, the clone density dropped due to the significant increase in the total number of tests, whilst the number of clone fragments stayed the same. We found that these two examples showcase the two common causes for these points of interest: either a large part of the code base is removed, such as a deprecated API in the Apache APR project, or a large amount of code is added, such as the set of hidden internal test in the Google Guave dataset. We conclude that the clone density in test code is much more sensitive to such large changes in the codebase, while clone density in production code is far more stable.

> The amount of duplication in test code is significantly higher than in production **at every point during development**. Moreover, clone density in test code is much more sensitive to changes in the code base, while clone density in production code is far more stable.

[3] Commit 461f265f2521781c2aa031334db587c9bcbb432a: *"Remove all uses of the apr_lock.h API from the tests."*
[4] Commit 31978328af5af66bc67bcbdda12b20710c2cfcad: *"Run "normal" GWT tests in public Guava."*

7.5.2 RQ1 Discussion

Our results show that duplication in test code is consistently higher throughout development, indicating that duplication is inherent to test code. We identified two main reasons why this is the case. First, the consistent structure of unit test code (the *setup-stimulate-verify-teardown* (S-S-V-T) cycle) induces a template which surfaces as clones [11]. Second, clones in production code can induce a multitude of clones in test code, as multiple test cases for duplicated functionality will themselves also be duplicated.

This will impact practitioners, as these inherent clones either can not or should not be refactored. There is no need to refactor test clones caused by the S-S-V-T cycle, as these can be considered as a particular instance of templating, and therefore should not be considered harmful [76]. On the other hand, test clones caused by duplicated functionality in the production code will be difficult to refactor, as both versions of the duplicated functionality need to be tested separately. In order to remove these test clones, the duplicated functionality in the production code will have to be refactored, not the test code itself. After the production code is refactored, then the test clones will disappear as a result.

We conclude that refactoring efforts need to be prioritized on production code, while duplicated test code can be used as an indication for which functionality needs to be refactored. The latter is especially true for purely semantic production clones, which are difficult to detect. As such, clone refactoring and maintanance tools that want to capitalize on this, will need to be able to ignore test duplication caused by the (S-S-V-T) cycle, whilst still detecting test clones caused by duplicated functionality in the production code.

7.5.3 RQ2: *Do Clones in Test Code Live Longer Compared to Clones in Production Code?*

Table 7.3 shows an overview of the number of alive and dead genealogies. We can see that test clones in C projects are likely to survive, as between 58% and 73% of test clones are alive. Production clones in C projects on the other hand are less likely to survive, with only between 24% and 58% of production clones still alive. In the Java projects, test clones also seem less likely to survive, with only between 19% and 49% of test clones alive. Production clones in Java projects are more similar to those in C projects, with 26% to 65% of production clones surviving. Overall, when looking at the total numbers, we can see that there the survivability of test clones is similar to that of production clones.

For a detailed overview of the age of clones, we provide a plot of the cumulative distribution for all k-volatile genealogies in Figure 7.4. Note that we omit the apache APR and libsodium graphs, as these projects only contain 17 and 4 dead test geneologies

Table 7.3: Clone Genealogies Statistics

Name		**Alive Genealogies**	**Dead Genealogies**
Google Guava	Prod	78 (26.99%)	211 (73.01%)
	Test	313 (31.78%)	672 (68.22%)
Apache Math	Prod	216 (35.53%)	392 (64.47%)
	Test	384 (33.54%)	761 (66.46%)
Apache Avro	Prod	56 (65.12%)	30 (34.88%)
	Test	118 (49.17%)	122 (50.83%)
Eclipse vert.x	Prod	52 (41.27%)	74 (58.73%)
	Test	29 (19.08%)	123 (80.92%)
Cairo	Prod	265 (24.91%)	799 (75.09%)
	Test	93 (67.39%)	45 (32.61%)
Apache APR	Prod	67 (46.53%)	77 (53.47%)
	Test	24 (58.54%)	17 (41.46%)
libarchive	Prod	90 (58.06%)	65 (41.94%)
	Test	82 (63.08%)	48 (36.92%)
libsodium	Prod	54 (49.54%)	55 (50.46%)
	Test	11 (73.33%)	4 (26.67%)
Total	Production	878 (34.02%)	1703 (66.98%)
	Test	1054 (37.03%)	1792 (62.97%)

respectively, which is too small a sample size. The graphs can be interpreted as follows. When considering the X-axis, the graph can be interpreted as "what percentage of clones has an age of X or lower". When considering the Y-axis, the graph can be interpreted as "at what age has Y% of clones disappeared". In our discussion, we will be using the latter interpretation.

When inspecting Figure 7.4, it is interesting to observe that the cumulative distribution approximates a logarithmic function for both production and test code. This means that most clones disappear relatively quickly, yet the oldest clones live significantly longer. Let us consider the Apache dataset for example. We can see that 90% of the clones have disappeared before reaching an age of 3,000 commits, yet the oldest clone reaches an age twice as high of 6,000 commits. Because of this, we will split our analysis in two parts. First, we will look at the short-lived clones by focussing on the first 50% of genealogies. Then, we will look at the long-lived clones, where we focus on the oldest 10% genealogies.

When looking at the short-lived clones, we can see that clones in production code generally live longer than clones in test code. For example, in the Google Guava project, 50% of all genealogies in test code have disappeared within 500 commits, while at that age only 40% of production genealogies have disappeared. Only after 1000 commits have 50%

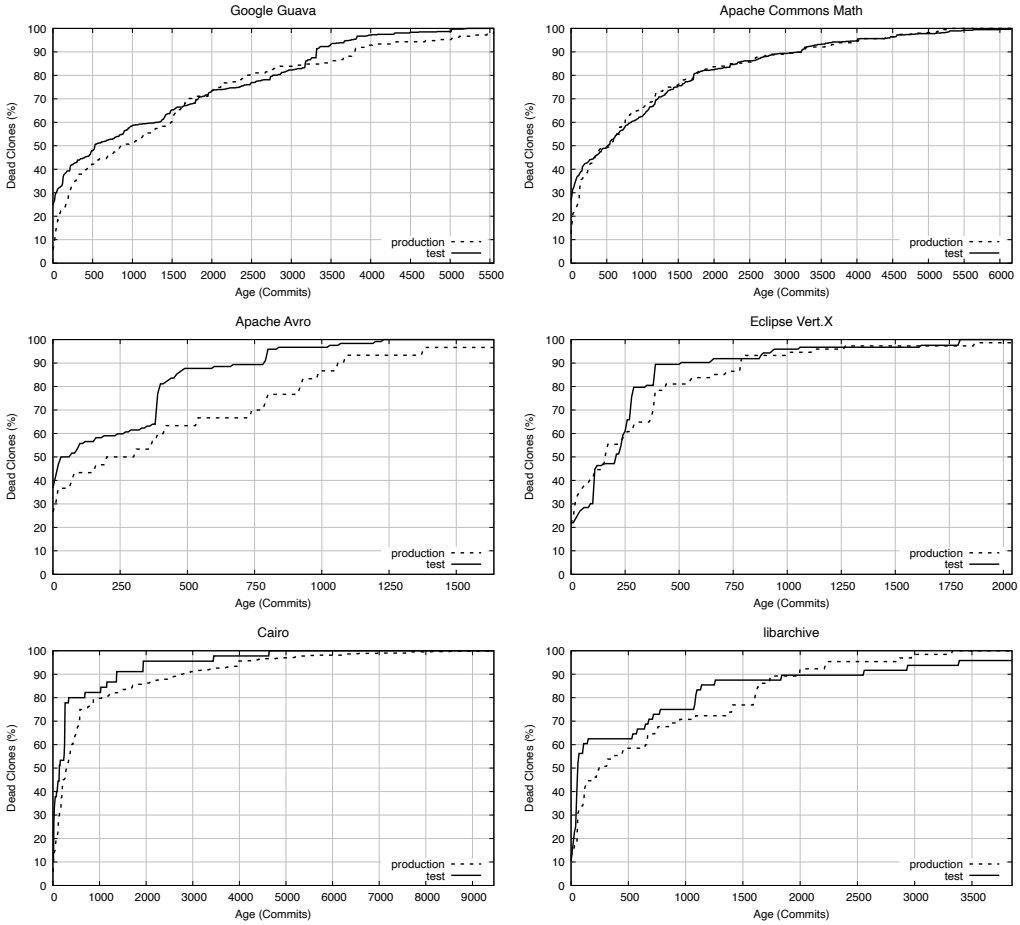

Figure 7.4: Cumulative distribution of k-volatile dead clones.

of all genealogies in production code disappeared. While the difference is less notable in the other datasets, such as in the Apache Commons Math library or the Ecliplse Vert.X project, most short-lived clone genealogies live longer in production code than in test code.

While long-lived test clones still live longer than long-lived production clones in the Apacha Avro and Cairo projects, and to a lesser extent in the Google Guava project, this is not the case for the other projects. In the Eclipse Vert.X project, the age of long-lived clones in production and test code becomes similar when considering the last 10%. In the libarchive project, the 10% oldest test clones actually live longer than their production counterpart. We therefore cannot draw a conclusion for long-lived clones when comparing test and production code.

We omitted alive genealogies from the previous analysis as it is impossible to know how long these clones will live. However, if there are many long-lived clones that are still alive, omitting these clones may skew the results. Therefore, in order to validate our previous observations, we also created the graphs including alive clones in Figure 7.5. It is important to note that we cannot use these graphs to draw conclusions about short-lived clones, as a part of them are still alive and in reality will end up living longer. We can, however, make observations about long-lived clones.

When comparing Figure 7.5 with Figure 7.4, we observe that the differences of long-lived clones between production code and test code have changed. Now, when we include alive genealogies, long-lived clones in production code live longer compared to those in test code, similar to what we concluded for short-lived clones previously. The one exception is the Cairo project, where 90% of production clones disappear before reaching an age of 5,000 commits, yet only 70% of test clones have disappeared by then. Yet, every other project confirms that our conclusion about short-lived clones also holds for long-lived clones.

> Overall, the survivability of test clones is similar to that of production clones. However, when clones do not survive, they disappear sooner in test code than in production code

7.5.4 RQ2 Discussion

Our results show that test clones are just as likely to survive as production clones, as both end up with around one third of all clones alive. This means that two thirds of all clones disappeared during the development process, either as a consequence of refactoring efforts or by chance. Our results show that these disappeared clones live shorter in test code compared to production code, indicating that they likely disappear naturally during the development process.

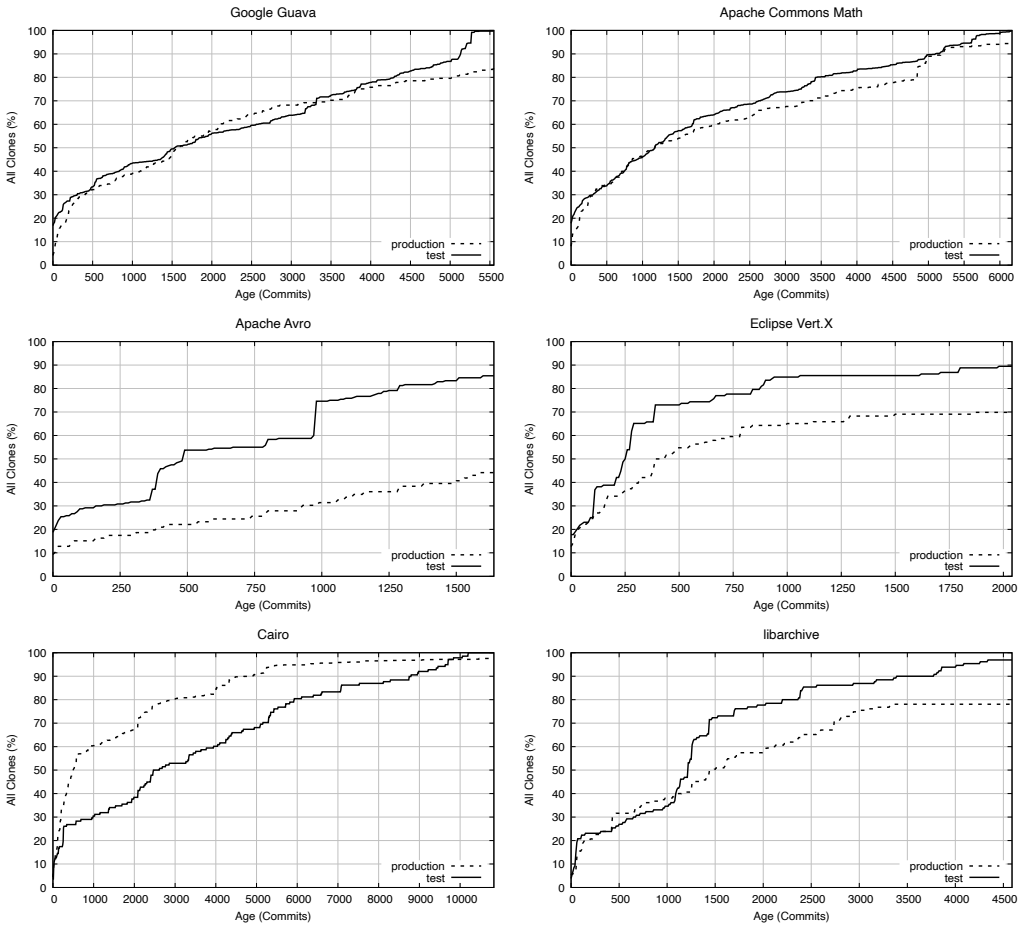

Figure 7.5: Cumulative distribution of all k-volatile clones.

Table 7.4: Clone Lineage Stability

Name		Consistent	Same	Inconsistent
Google Guava	Prod	16.96%	58.13%	24.91%
	Test	29.64%	46.80%	23.55%
Apache Math	Prod	24.01%	37.66%	38.32%
	Test	25.41%	33.19%	41.40%
Apache Avro	Prod	12.79%	63.95%	23.26%
	Test	28.75%	45.00%	26.25%
Eclipse vert.x	Prod	13.49%	72.22%	14.29%
	Test	25.00%	55.92%	19.08%
Cairo	Prod	40.88%	43.14%	15.98%
	Test	28.99%	50.00%	21.01%
Apache APR	Prod	19.44%	67.36%	13.19%
	Test	41.46%	34.15%	24.39%
libarchive	Prod	16.77%	62.58%	20.65%
	Test	33.08%	34.62%	32.31%
libsodium	Prod	28.44%	59.63%	11.93%
	Test	20.00%	13.33%	66.67%

This confirms our previous conclusion that refactoring efforts need to be prioritized on production code, where more longer-living clones reside. However, even though test clones do not necessarily need to be refactored, they still need to be maintained. A change to the functionality under test will require all related duplicate tests to be adjusted as well. As such, practitioners should take this into account during effort estimation, as continuous tracking of a higher number of short-lived test clones will require more maintenance effort.

7.5.5 RQ3: *How Do Clones in Production Code and Test Code Evolve According to the Evolution and Change Patterns?*

Table 7.4 shows for each dataset the number of lineages that never changed, only changed consistently, and changed inconsistently at least once. We can see that in almost all projects, around 60% of production clones never change after being introduced. Only in the Apache Math library and the Cairo project, this number is lower, with around 40% of clones never being changed after introduction. This confirms findings from previous research, namely that cloned code is generally stable [96, 103]. However, when considering test code, we can see that the percentage of unchanged lineages is consistently lower than their production counterparts. A larger percentage of clones in test code is changed after introduction compared to production code. Or in other words, test clones are less stable than production clones.

Table 7.5: Evolution and Change Patterns of Clone Versions

Name		Change Pattern		Evolution Pattern	
		Consistent	Inconsistent	Add	Subtract
Google Guava	Prod	81.32%	18.68%	50.24%	49.76%
	Test	81.67%	18.33%	49.62%	50.38%
Apache Math	Prod	75.83%	24.17%	49.76%	50.24%
	Test	70.82%	29.18%	50.35%	49.65%
Apache Avro	Prod	85.59%	14.41%	49.91%	50.09%
	Test	70.81%	29.19%	49.13%	50.87%
Eclipse vert.x	Prod	80.65%	19.35%	50.16%	49.84%
	Test	84.06%	15.94%	48.88%	51.12%
Cairo	Prod	79.79%	20.21%	53.81%	46.19%
	Test	67.15%	32.85%	81.94%	18.06%
Apache APR	Prod	77.29%	22.71%	48.00%	52.00%
	Test	68.85%	31.15%	58.33%	41.67%
libarchive	Prod	60.77%	39.23%	80.00%	20.00%
	Test	76.35%	23.65%	41.67%	58.33%
libsodium	Prod	80.49%	19.51%	54.55%	45.45%
	Test	62.75%	37.25%	31.58%	68.42%

When test lineages change, they change consistently more often than inconsistently. The Apache Math library is again an exception here, with a larger percentage of lineages changed inconsistently. The libsodium project also sees a significant number of inconsistently changed test lineages, yet these numbers might be skewed as there are only 15 lineages in the project's test code. This is confirmed by Table 7.5, where 62% of clone versions are in fact changed consistently. We can conclude that often, when one test is changed, all duplicated tests are changed as well.

When production lineages change, they change inconsistently more often than consistently. Although there are some exceptions here as well, most notably the Cairo project. When comparing production and test lineages, we can see that test lineages are changed consistently more often than production lineages, and they are also changed inconsistently more often as well. This is of course due to the smaller number of unchanged lineages in test code, increasing the number of changed lineages both consistently and inconsistently.

Table 7.5 shows how often each change pattern and evolution pattern occurs in total. Note that, contrary to Table 7.4, we now consider all clone versions instead of just the clone lineages, meaning that if the same clone is changed multiple times, each change is recorded. Also note that we omit the "Same" category in this table, as the number of

times any clone version is unaffected by any commit does not provide meaningful data. We can see that, for both clones in production code and clones in test code, changes are most often consistent.

When comparing Table 7.4 and Table 7.5, the change patterns differ greatly when considering them on a version granularity compared to a lineage granularity. More specifically, the ratio of consistent and inconsistent clone lineages does not match the ratio of consistent and inconsistent clone versions, as there are many more consistently changed versions than consistent lineages. This is caused by two reasons. Firstly, both consistent and inconsistent lineages contain 5 consistent changes on average, while inconsistent lineages only contain 2 inconsistent changes on average. Secondly, a handful of highly unstable lineages contain hundreds of consistent changes. The maximum we found was one lineage from the Google Guava dataset, which changed consistently 2,066 times. As a result, the percentage of consistently changed clone versions is significantly higher than the percentage of consistently changed clone lineages.

When considering the evolution patterns, there again seems no inherent difference between production clones and test clones. For both, the number of versions where a clone fragment is added is similar to that where a clone fragment is removed.

Clones in test code are less stable than those in production code, yet they are more likely to change consistently than inconsistently.

7.5.6 RQ3 Discussion

Our results show that clones in test code not only change more often than clones in production code, but changes are also applied consistently more often. This is to be expected, since duplication in test code mostly stems from (A) multiple similar unit tests testing a certain functionality, or (B) multiple tests testing the same functionality for duplicated production code; with both types of clone inherently requiring consistent change. While this is again in line with the use of the S-S-V-T template, it does raise an interesting problem. As changes need to be consistently applied to all duplicated yet adapted test cases, it becomes important for practitioners to be able to find and track these duplicated tests. On a small scale, this can be achieved with naming conventions, documentation, and grouping duplicated tests together in files. On a larger scale, as consistent maintenance and tracking of test clones becomes more important yet difficult, the need for special purpose clone tracking and management tools increases. Such tools may exploit the presence of the S-S-V-T template to provide actionable recommendations.

7.6 THREATS TO VALIDITY

7.6.1 Internal Validity

We are aware of three main threats to the internal validity of our study. The first threat lies in the way we determine whether a clone changed consistently or inconsistently between two versions of the system. We do this by comparing the hash of the normalized code in both versions in order to detect change. While this method does accurately detect change in the code, we have no information about what that change was. We assume that if both fragments of a clone change in the same version, that this change was consistent. However, it is possible that a different (inconsistent) change was made to both fragments in the same version, which would result in a false positives in our classification.

A second threat to internal validity comes from tracking the clones throughout development. We track clones with their previous version based on the name of the file and function. However, if a file or function was renamed during development, we are not able to track the containing clones. This would result in a clone lineage being split in two.

The third threat lies in the fact that we only analyse every tenth commit. Analysing every commit is not feasible due to hardware limitations, so we experimentaly decided on a 10-commit interval. It is possible that a clone is changed multiple times in that 10-commit interval, in which case we would only detect one change. This would mean that our results are, in general, a slight underestimation of reality.

All of these threats stem from the limitations of automation. In order to remove these threats, we would need to manually analyse every clone in every version, which would not be possible on this scale. Therefore, we are confident that the benefit of automation, namely the large scale of our study, outweighs the potential of a small number of false positives. We are also confident that a small number of false positives would not alter our conclusions, as these are based on a significantly large amount of data.

7.6.2 External Validity

We are aware of two main threats to the external validity of our study. The first threat lies in the way we detect clones. We use two clone detector, namely NiCad and iClones. While these are reliable, time-tested, and commonly-used clone detectors, it is possible that there are additional clones in the dataset that were not detected. However, since we used the same clone detectors on both the production and the test datasets, and since we did find an number of clones in both that is in line with previous research, we are confident that our comparison is fair and correct, and that our conclusions hold.

A second threat comes from our dataset. In our evaluation, we use four open-source Java projects and four open-source C projects. A threat to external validity is that these projects, and the clones within these projects, are not representative of all clones and/or code bases. To minimize this threat, we chose the projects in our dataset such that they vary in size, type, complexity, and language. We encourage future research to confirm our findings by adding more datasets and clone detection tools to our evaluation.

7.7 CONCLUSION

In this paper, we investigated the inherent differences between test and production clones through the lens of *"clone genealogies"*. Clone genealogies are the accepted way of studying the evolution of clones in academic literature and has led to many interesting insights in the past [92, 93, 96, 106, 107]. Nevertheless, these insights were always based on studying clones in production code and so far did not explicitly consider what happens in test code, where code clones are known to occur far more often [10, 11, 58].

We collected data on 5,427 clone genealogies from eight open-source projects; four written in Java and four written in C. We studied the differences between these genealogies as they appear in production code and in test code using three research questions.

RQ1: *How do the clone density for production code and test code evolve throughout development?* We observed that at every point during development the amount of duplication in test code is significantly higher than in production code. We identified two main reasons why this is the case. First, the consistent structure of unit test code (the *setup-stimulate-verify-teardown* (S-S-V-T) cycle [41]) induces a template which surfaces as clones [11]. Second, clones in production code can induce a multitude of clones in test code, as multiple test cases for duplicated functionality will themselves also be duplicated. We conclude that duplication is inherent to test code, and that refactoring efforts need to be prioritized on production code while duplicated test code can serve as an indication for which functionality needs to be refactored.

RQ2: *Do clones in test code live longer compared to clones in production code?* Overall the survivability of test clones is similar to that of production clones. However, when clones do not survive, they disappear sooner in test code than in production code. We conclude that practitioners should take this into account during effort estimation, as continuous tracking of a higher number of short-lived test clones will require more maintenance effort.

RQ3: *How do clones in production code and test code evolve according to the evolution and change patterns?* We found that clones in test code are less stable than those in production code, yet they are more likely to change consistently than inconsistently. This raises an interesting problem, as changes need to be consistently applied to all duplicated yet adapted test cases, it becomes important for practitioners to be able to find and track these duplicated tests. We conclude that, as consistent maintenance and tracking of test clones becomes more important and more difficult, the need for clone detection, tracking, and management tools increases.

These observations confirm previous research that test code is a rich source for studying clones and warrants further investigation. In particular we see following avenues for further research (FR).

FR1: *Pro-active recommendations for refactoring:* While there is a consensus that not all clones should be considered harmful, clones that induce downstream clones in test code represent a prime refactoring opportunity.

$\Longrightarrow$ Clone tracking and managing tools should more deeply study the phenomenon of production code clones causing an avalanche of clones in test code and provide actionable suggestions thereon.

FR2: *Missing abstractions:* Many of the short-lived clones in test code are changed consistently, which appears to be a case of *"shotgun surgery"* [40]. This suggests that today's unit testing frameworks miss certain abstractions to express typical coding patterns.

$\Longrightarrow$ Developers involved in unit testing frameworks should scrutinise such consistently changing long lived clones to see where the S-S-V-T cycle could be refined.

FR3: *Revisit benchmarks for a threat to construct validity:* All of our research illustrates that clones in test code should be treated separately from clones in production code. However, the commonly used benchmarks for code clones (most notably [60, 77]) don't make that distinction, which may be a threat to construct validity.

$\Longrightarrow$ Benchmark creators should refine their datasets and add meta-data to distinguish between a clone that appears in production code and one that appears in test code.

$\Longrightarrow$ Researchers relying on these benchmarks should reconsider their conclusions and see whether this threat to validity affects their results.

Chapter 8

Conclusion

In this thesis, we investigated code duplication in test code based on two research questions.

RQ1: *Can we exploit the structure of test code to detect semantic code clones?*
In Chapters 3 to 5, we presented a new technique to detect semantic code clones in test code. We first provided a theoretical model that defines test behaviour, in the form of Test Behaviour Trees. We showed that it is feasible to use our theoretical model by implementing T-CORE, a tool that uses symbolic execution to create the Test Behaviour Tree for a unit test in order to detect changes in test behaviour. Then, we proposed that T-CORE could be used to detect semantic code clones in test code by comparing the Test Behaviour Trees of unit tests. We demonstrated on the Apache Commons Math Library's test suite that our approach detects 755 clone pairs with a precision of 98%. We also showed that 259 of the 755 detected clone pairs are type-IV clones, i.e. semantic clones. This confirms that it is both feasible and worthwhile to investigate semantic clones in test code.

RQ2: *Should test code duplication be considered independently of production code duplication?*
In Chapters 6 to 7, we performed a series of empirical studies into test code duplication, comparing it with production code duplication. We observed that, at every point during development, the amount of duplication in test code is significantly higher than in production code. This increase can be attributed to significantly more occurrences of Type II and Type III clones in test code. In other words, many tests are duplicated multiple times, each time only slightly modified. We identified two main reasons why this is the case. First, the consistent structure of unit test code (the *setup-stimulate-verify-teardown* (S-S-V-T) cycle [41]) induces a template which surfaces as clones [11]. Second, clones in

production code can induce a multitude of clones in test code, as multiple test cases for duplicated functionality will themselves also be duplicated. We conclude that duplication is inherent to test code, and that refactoring efforts need to be prioritized on production code while duplicated test code can serve as an indication for which functionality needs to be refactored.

We also found that, while overall the survivability of test clones is similar to that of production clones, code clones disappear sooner in test code than in production code. Practitioners should take this into account during effort estimation, as continuous tracking of a higher amount of short-lived test clones will require more maintenance effort. Moreover, clones in test code are less stable than those in production code, yet they are more likely to change consistently than inconsistently. This raises an interesting problem, as changes need to be consistently applied to all duplicated yet adapted test cases, it becomes important for practitioners to be able to find and track these duplicated tests. We conclude that, as consistent maintenance and tracking of test clones becomes more important and more difficult, the need for clone detection, tracking, and management tools increases.

These observations confirm that test code is a rich source for studying clones and warrants further investigation.

Bibliography

[1] Elfriede Dustin. *Effective Software Testing: 50 Ways to Improve Your Software Testing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201794292. (Cited on pages 1, 10, and 22).

[2] Milind G Limaye. *Software testing*. Tata McGraw-Hill Education, 2009. (Cited on pages 1 and 66).

[3] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007. (Cited on pages 2, 42, and 66).

[4] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2009. (Cited on pages 2, 42, and 66).

[5] Huiqing Li, Adam Lindberg, Andreas Schumacher, and Simon Thompson. Improving your test code with wrangler. Technical report, School of Computing, University of Kent, 2009. (Cited on pages 2, 42, 51, and 66).

[6] Rainer Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007. (Cited on pages 2, 44, and 69).

[7] Brent van Bladel and Serge Demeyer. Test refactoring: a research agenda. In *Postproceedings of the Tenth Seminar on Advanced Techniques and Tools for Software Evolution*. CEUR, 2017. (Cited on pages 4, 23, and 32).

[8] Brent van Bladel and Serge Demeyer. Test behaviour detection as a test refactoring safety. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 22–25. ACM, 2018. (Cited on pages 4 and 32).

[9] Brent van Bladel and Serge Demeyer. A novel approach for detecting type-iv clones in test code. In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pages 8–12. IEEE, 2019. (Cited on pages 4, 46, and 49).

[10] Brent van Bladel and Serge Demeyer. A comparative study of test code clones and production code clones. *Journal of Systems and Software*, 176:110940, 2021. (Cited on pages 4, 66, 69, 74, 76, 78, and 88).

[11] Brent van Bladel and Serge Demeyer. Clone detection in test code: An empirical evaluation. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 492–500. IEEE, 2020. (Cited on pages 4, 42, 45, 50, 62, 66, 79, 88, and 91).

[12] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998. (Cited on pages 5, 44, 46, 50, and 74).

[13] Chanchal K Roy and James R Cordy. An empirical study of function clones in open source software. In *2008 15th Working Conference on Reverse Engineering*, pages 81–90. IEEE, 2008. (Cited on pages 5, 44, 50, and 74).

[14] Chanchal K Roy and James R Cordy. Near-miss function clones in open source software: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):165–189, 2010. (Cited on pages 5, 44, 50, and 74).

[15] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2009. (Cited on page 10).

[16] A Van Deursen, L Moonen, A van den Bergh, and G Kok. Refactoring test code. In *2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pages 92–95. University of Cagliary, 2001. (Cited on pages 10, 13, 22, and 30).

[17] Arie Van Deursen and Leon Moonen. The video store revisited–thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76. Citeseer, 2002. (Cited on pages 10 and 22).

[18] Jens Uwe Pipka. Refactoring in a "test first"-world. In *Proc. Third Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng*, 2002. (Cited on pages 10 and 22).

[19] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007. (Cited on pages 10 and 13).

[20] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 322–331. IEEE, 2013. (Cited on pages 10, 12, and 13).

[21] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 391–400. IEEE, 2006. (Cited on pages 11 and 22).

[22] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 56–65. IEEE, 2012. (Cited on pages 11, 22, 42, and 45).

[23] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015. (Cited on pages 11 and 22).

[24] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15. ACM, 2016. (Cited on page 11).

[25] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 450–457. IEEE, 2011. (Cited on page 11).

[26] Manuel Breugelmans and Bart Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008. (Cited on pages 11 and 13).

[27] Ali Parsai, Alessandro Murgia, Quinten David Soetens, and Serge Demeyer. Mutation testing as a safety net for test code refactoring. In *Scientific Workshop Proceedings of the XP2015*, page 8. ACM, 2015. (Cited on pages 12 and 23).

[28] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997. (Cited on page 22).

[29] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001. (Cited on page 22).

[30] Vahid Garousi et al. Smells in software test code: a survey of knowledge in industry and academia. *Journal of Systems and Software*, 2017. (Cited on page 23).

[31] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM. doi: 10.1145/800191.805647. URL `http://doi.acm.org/10.1145/800191.805647`. (Cited on page 24).

[32] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL `http://doi.acm.org/10.1145/360248.360252`. (Cited on page 24).

[33] Robert Neil Faiman Jr. Method of constructing a constant-folding mechanism in a multilanguage optimizing compiler, November 10 1998. US Patent 5,836,014. (Cited on pages 24 and 33).

[34] Jean-Paul Tremblay and Paul G Sorenson. *Theory and Practice of Compiler Writing*. McGraw-Hill, Inc., 1985. (Cited on page 24).

[35] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. (Cited on page 24).

[36] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc., 1st edition, 2014. ISBN 1449372627, 9781449372620. (Cited on page 25).

[37] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*, ICSE 2013, pages 422–431, May 2013. (Cited on page 27).

[38] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001. (Cited on page 30).

[39] Outi Salo and Pekka Abrahamsson. Agile methods in European embedded software development organisations: a survey on the actual use and usefulness of Extreme Programming and SCRUM. *IET software*, 2(1):58–64, 2008. (Cited on page 30).

[40] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999. (Cited on pages 30 and 89).

[41] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007. (Cited on pages 30, 42, 51, 52, 63, 66, 88, and 91).

[42] 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)*, pages 482–489, July 2018. doi: 10.1109/QRS.2018.00061. (Cited on page 30).

[43] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. Technical report, Technical Report 541, Queen's University at Kingston, 2007. (Cited on pages 30, 31, 34, 38, and 53).

[44] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International static analysis symposium*, pages 40–56. Springer, 2001. (Cited on pages 31 and 46).

[45] Jens Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309. IEEE, 2001. (Cited on pages 31 and 46).

[46] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330. ACM, 2008. (Cited on page 31).

[47] Daniel E Krutz and Emad Shihab. CCCD: Concolic Code Clone Detection. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 489–490. IEEE, 2013. (Cited on page 31).

[48] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. MeCC: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310. ACM, 2011. (Cited on pages 31 and 46).

[49] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92. ACM, 2009. (Cited on pages 31 and 46).

[50] Thierry Lavoie, Mathieu Mérineau, Ettore Merlo, and Pascal Potvin. A case study of TTCN-3 test scripts clone analysis in an industrial telecommunication setting. *Information and Software Technology*, 87:32–45, 2017. (Cited on page 31).

[51] James R Cordy and Chanchal K Roy. The NiCad clone detector. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 219–220. IEEE, 2011. (Cited on pages 34 and 47).

[52] Kent Beck. *Test-driven Development: By Example*. Kent Beck signature book. Addison-Wesley, 2003. ISBN 9780321146533. (Cited on page 42).

[53] Grady Booch. *Object Oriented Design: With Applications*. Benjamin/Cummings Pub., 1991. ISBN 9780805300918. (Cited on page 42).

[54] Martin Fowler and Matthew Foemmel. Continuous integration. Technical report, Thoughtworks, 2006. (Cited on page 42).

[55] John D. McGregor. Test early, test often. *Journal of Object Technology*, 6(4):7–14, May 2007. ISSN 1660-1769. doi: 10.5381/jot.2007.6.4.c1. URL `http://dx.doi.org/10.5381/jot.2007.6.4.c1`. (column). (Cited on page 42).

[56] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. (Cited on page 42).

[57] Vahid Garousi and Baris Kucuk. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52 – 81, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.12.013. (Cited on pages 42 and 45).

[58] Wafa Hasanain, Yvan Labiche, and Sigrid Eldh. An analysis of complex industrial test code using clone analysis. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 482–489. IEEE, 2018. (Cited on pages 42, 45, 50, 66, and 88).

[59] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009. (Cited on pages 44, 46, 47, 69, and 71).

[60] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, 2014. (Cited on pages 44, 52, 62, and 89).

[61] Chanchal K Roy and James R Cordy. Benchmarks for software clone detection: A ten-year retrospective. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 26–37. IEEE, 2018. (Cited on pages 44, 52, and 69).

[62] Sarah J Clarke and Peter Willett. Estimating the recall performance of web search engines. In *Aslib proceedings*. MCB UP Ltd, 1997. (Cited on pages 44 and 52).

[63] Nildo Silva Junior, Larissa Rocha, Luana Almeida Martins, and Ivan Machado. A survey on test practitioners' awareness of test smells. *arXiv preprint arXiv:2003.05613*, 2020. (Cited on page 45).

[64] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, 2015. (Cited on page 45).

[65] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016. (Cited on page 46).

[66] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina 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*, pages 354–365, 2018. (Cited on page 46).

[67] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007. (Cited on page 46).

[68] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002. (Cited on page 46).

[69] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016. (Cited on page 46).

[70] Jeffrey Svajlenko and Chanchal 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)*, pages 177–179. IEEE, 2017. (Cited on page 46).

[71] Chanchal K Roy and James 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*, pages 172–181. IEEE, 2008. (Cited on page 47).

[72] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262. IEEE, 2006. (Cited on page 47).

[73] Nils Göde and Rainer Koschke. Incremental clone detection. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 219–228. IEEE, 2009. (Cited on page 47).

[74] Paul Jaccard. Nouvelles recherches sur la distribution florale. *Bull. Soc. Vaud. Sci. Nat.*, 44:223–270, 1908. (Cited on page 51).

[75] Seung-Seok Choi, Sung-Hyuk Cha, and Charles C Tappert. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, 8(1):43–48, 2010. (Cited on page 51).

[76] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645, 2008. (Cited on pages 52 and 79).

[77] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591, 2007. (Cited on pages 52, 62, and 89).

[78] Miryung Kim, Vibha Sazawal, David Notkin, and Gail 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)*, page 187–196, New York, NY, USA, 2005. ACM. ISBN 1595930140. doi: 10.1145/1081706.1081737. (Cited on page 61).

[79] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 170–178, 2007. (Cited on page 62).

[80] S. Kawuma, J. Businge, and E. Bainomugisha. Can we find stable alternatives for unstable eclipse interfaces? In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016. (Cited on page 62).

[81] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *Journal of Systems and Software*, 157, 2019. (Cited on page 62).

[82] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. Automatic test improvement with dspot: a study with ten mature open-source projects. *Empirical Software Engineering, Springer Verlag*, 2019. (Cited on page 62).

[83] Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. Test amplification in the pharo smalltalk ecosystem. In *Proceedings IWST 2019* (*International Workshop on Smalltalk Technologies*). ESUG, 2019. (Cited on page 62).

[84] Mitchel Pyl, Brent van Bladel, and Serge Demeyer. An empirical study on accidental cross-project code clones. In *2020 IEEE 14th International Workshop on Software Clones* (*IWSC*), pages 33–37. IEEE, 2020. (Cited on page 62).

[85] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings ISSTA 2015* (*International Symposium on Software Testing and Analysis*), page 257–269, New York, NY, USA, 2015. ACM. ISBN 9781450336208. doi: 10.1145/2771783.2771796. (Cited on page 62).

[86] T. Zhang and M. Kim. Automated transplantation and differential testing for clones. In *Proceedings ICSE 2017* (*IEEE/ACM 39th International Conference on Software Engineering*), pages 665–676, 2017. (Cited on page 62).

[87] C. K. Roy, M. F. Zibran, and 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*), pages 18–33, 2014. (Cited on page 63).

[88] Ekwa Duala-Ekoko and Martin P. Robillard. Clonetracker: Tool support for code clone management. In *Proceedings ICSE 2008* (*30th International Conference on Software Engineering*), ICSE '08, page 843–846, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368218. (Cited on page 63).

[89] G. Zhang, X. Peng, Z. Xing, Shihai Jiang, Hai Wang, and 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*), pages 497–507, 2013. (Cited on page 63).

[90] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano. Applying clone change notification system into an industrial development process. In *Proceedings ICPC (21st International Conference on Program Comprehension)*, pages 199–206, 2013. (Cited on page 63).

[91] S. Tokui, N. Yoshida, E. Choi, and 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)*, pages 642–646, 2020. (Cited on page 63).

[92] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, 2005. (Cited on pages 66, 67, 69, 73, 75, and 88).

[93] Miryung Kim and David Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005. (Cited on pages 70 and 88).

[94] Jens Krinke. Changes to code clones in evolving software. *Softwaretechnik-Trends*, 27(2), 2007. (Cited on page 70).

[95] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *14th working conference on reverse engineering (WCRE 2007)*, pages 170–178. IEEE, 2007. (Cited on page 70).

[96] Jens Krinke. Is cloned code more stable than non-cloned code? In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 57–66. IEEE, 2008. (Cited on pages 70, 84, and 88).

[97] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. Clone smells in software evolution. In *2007 IEEE International Conference on Software Maintenance*, pages 24–33. IEEE, 2007. (Cited on pages 70 and 73).

[98] Ekwa Duala-Ekoko and Martin P Robillard. Tracking code clones in evolving software. In *29th International Conference on Software Engineering (ICSE'07)*, pages 158–167. IEEE, 2007. (Cited on page 70).

[99] Jan Harder and Nils Göde. Modeling clone evolution. *Proc. IWSC*, pages 17–21, 2009. (Cited on page 70).

[100] Nils Göde and Marcus Rausch. Clone evolution revisited. *Softwaretechnik-Trends*, 30(2):60–61, 2010. (Cited on page 70).

[101] Tibor Bakota. Tracking the evolution of code clones. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 86–98. Springer, 2011. (Cited on page 70).

[102] Jens Krinke. Is cloned code older than non-cloned code? In *Proceedings of the 5th International Workshop on Software Clones*, pages 28–33, 2011. (Cited on page 70).

[103] Nils Göde and Jan Harder. Clone stability. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 65–74. IEEE, 2011. (Cited on pages 70 and 84).

[104] Ripon K Saha, Muhammad Asaduzzaman, Minhaz F Zibran, Chanchal K Roy, and Kevin A Schneider. Evaluating code clone genealogies at release level: An empirical study. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 87–96. IEEE, 2010. (Cited on page 70).

[105] Nils Göde and Jan Harder. Oops!... i changed it again. In *Proceedings of the 5th international workshop on software clones*, pages 14–20, 2011. (Cited on page 70).

[106] Nils Göde and Rainer Koschke. Frequency and risks of changes to clones. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 311–320, 2011. (Cited on pages 70 and 88).

[107] Nicolas Bettenburg, Weiyi Shang, Walid M Ibrahim, Bram Adams, Ying Zou, and Ahmed E Hassan. An empirical study on inconsistent changes to code clones at the release level. *Science of Computer Programming*, 77(6):760–776, 2012. (Cited on pages 70 and 88).

[108] Ripon K Saha, Chanchal K Roy, and Kevin A Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 293–302. IEEE, 2011. (Cited on page 70).

[109] Meng Ci, Xiao-hong Su, Tian-tian Wang, and Pei-jun Ma. A new clone group mapping algorithm for extracting clone genealogy on multi-version software. In *2013 Third International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 848–853. IEEE, 2013. (Cited on page 70).

[110] Nils Göde and Rainer Koschke. Studying clone evolution using incremental clone detection. *Journal of Software: Evolution and Process*, 25(2):165–192, 2013. (Cited on page 70).

[111] Ripon K Saha, Chanchal K Roy, and Kevin A Schneider. gcad: A near-miss clone genealogy extractor to support clone evolution analysis. In *2013 IEEE International Conference on Software Maintenance*, pages 488–491. IEEE, 2013. (Cited on page 70).

[112] Liliane Barbour, Foutse Khomh, and Ying Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Software: Evolution and Process*, 25(11): 1139–1165, 2013. (Cited on page 70).

[113] Hsiao Hui Mui, Andy Zaidman, and Martin Pinzger. Studying late propagations in code clone evolution using software repository mining. *Electronic Communications of the EASST*, 63, 2014. (Cited on page 70).

[114] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Late propagation in near-miss clones: An empirical study. *Electronic Communications of the EASST*, 63, 2014. (Cited on page 70).

[115] Shuai Xie, Foutse Khomh, Ying Zou, and Iman Keivanloo. An empirical study on the fault-proneness of clone migration in clone genealogies. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 94–103. IEEE, 2014. (Cited on page 70).

[116] Liliane Barbour, Le An, Foutse Khomh, Ying Zou, and Shaohua Wang. An investigation of the fault-proneness of clone evolutionary patterns. *Software Quality Journal*, 26(4):1187–1222, 2018. (Cited on page 70).

[117] Manishankar Mondal, Banani Roy, Chanchal K Roy, and Kevin A Schneider. An empirical study on bug propagation through code cloning. *Journal of Systems and Software*, 158:110407, 2019. (Cited on page 70).

[118] Fanlong Zhang, Siau-Cheng Khoo, and Xiaohong Su. Predicting consistent clone change. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 353–364. IEEE, 2016. (Cited on page 70).