

How Multiple Developers Affect the Evolution of Code Clones

Jan Harder

Software Engineering Group
University of Bremen, Germany
harder@informatik.uni-bremen.de

Abstract—The use of copy and paste in programming causes redundant passages of source code. The effect such clones have on software quality and maintainability in particular has been subject to various studies in the recent past. Although negative effects could not be confirmed in general, a non-negligible number of situations where clones did cause problems has been found. Hence, there may be yet unknown influencing factors that cause these problems. One such factor may be the number of developers involved in the creation and maintenance of a clone. The interaction of multiple developers, unevenly distributed knowledge and communication deficiencies may lead to unwanted inconsistencies and bugs, when the clones are changed. This paper presents an empirical study on long-lived software systems, in which we analyze how many developers are involved in the maintenance exact clones and whether the number of developers affects the likelihood of inconsistent changes. Our results indicate that differences between single-author and multi-author clones exist. Nevertheless, we did not find multiple developers to be the cause of problematic changes to clones.

Keywords—code clones; code authorship; software evolution; software quality;

I. INTRODUCTION

The source code of software systems often contains redundant passages that resulted from copy-and-paste programming. There is a widespread belief that such code clones complicate software maintenance, because changes must be applied repeatedly to all copies. Furthermore, it is often assumed that failing to change all copies consistently may cause new bugs or incomplete bug removals.

These conjectures have been subject to various studies in the recent past. The results are diverse. On the one hand, higher maintenance activity—and therefore higher costs—as well as a general tendency to bugs could not be confirmed for cloned code. On the other hand, cases where cloning caused bugs have been identified. These results suggest that clones do not pose a threat to correctness or maintenance in general. Nevertheless, some clones do. As to now, no method to separate the ‘good’ from the ‘bad’ is known.

One possible factor that may complicate the maintenance of clones is the number of developers involved in creating and changing them. While a single developer may be able to keep track of the copies she or he made, copying someone else’s code without letting her or him know, could result in unintended inconsistent changes to the clones in the future. So far, possible authorship effects like these have rarely been researched. Balint and colleagues categorized changes to

clones through different authors in an explorational study [1]. They found inconsistent changes by different authors to be a reoccurring pattern and suggest to use the number of developers, who are involved in cloning, as a metric to prioritize clones in practice. Nevertheless, the frequency of such changes and whether the inconsistency was indeed unintended has not been evaluated, yet. Cai and Kim correlated the life-time of clones with different metrics and found that the number of developers has a strong positive correlation [2]. That is, clones, to which multiple developers contribute, remain in the system longer. It remains, however, unknown whether a long lifetime is positive (e.g., code maturity) or negative (e.g., difficult to remove).

Further indication that multiple developers may cause problems with clones is given by an observational field study on program comprehension by Roehm and colleagues [3]. They observed that developers tend to copy code from others instead of changing or extending it. Either because they do not fully understand the implementation or because they cannot foresee all consequences of changing the existing code fragment. When clones are created because of such uncertainty and communication is avoided, the question whether they lead to further problems arises. In a self-experiment we found it difficult to decide whether an inconsistent clone poses a problem, when the original and the copy were created by different developers [4]. It was particularly difficult to gather all required information from different people.

In this study, we analyze the relationship between code clones and code authorship in five subject systems. Our aim is to shed light on the aforementioned assumptions. To this end, we define the following research questions.

Question 1 — *How many developers are involved in the creation and maintenance of clones?*

Before the effects of multiple developers can be evaluated, we need to examine how often clones are indeed authored by more than one developer. We define a mechanism to detect the authorship of cloned code fragments and quantify how often multiple developers are involved in the creation and maintenance of clones.

Question 2 — *Do clones change more often when multiple developers are involved?*

We analyze whether clones that have multiple authors are more likely to change. An increased change frequency can imply higher maintenance effort. Each change entails the risk of an unintended inconsistency.

Question 3 — *Are inconsistent changes more likely if multiple developers are involved?*

We also analyze the change consistency. That is, whether the changes to a clone are applied consistently to all copies. Inconsistent changes can cause faulty program behavior. If a cloned bug is fixed inconsistently, the system remains in a defective state.

The above questions will be pursued statistically. While statistics will shed first light on the general relationship, manual inspection of the clones and the changes made to them is essential to understand the rationale behind the cloning. Possible problems, such as unintended inconsistencies and bugs, can only be safely assessed with manual inspections.

Question 4 — *Do multi-author clones cause unintended inconsistencies or bugs? Does the rationale behind single-author and multi-author cloning differ?*

Contribution. Our main contribution is the first empirical data on the effect multiple developers have on clones. We also provide first insights into the rationale of single-author and multi-author cloning. The results provide a basis for the decision whether the number of developers may be used to prioritize clone management activity. In order to detect code authorship, we also present a new method based on the metadata of `Subversion` and the source code. Authorship is analyzed on the level of tokens, which is more precise than other methods that have been proposed previously.

Outline. The remainder of this paper is organized as follows. In Section II we give an overview on the related research. Our authorship detection technique is explained in Section III. Section IV describes the setup of our case study, which is followed by the presentation of the results in Section V. In Section VI we discuss possible implications of our results. Possible threats to validity are covered in section VII. Section VIII concludes.

II. RELATED WORK

Code clones have been subject to research for more than two decades. Various methods and tools to detect clones in source code exist and have been applied in research and practice. The interpretation of the—often vast amount—of results and the management of the detected clones, are an active area of research. An overview is given in the surveys by Koschke [5] and Roy and colleagues [6].

A. Effects of Clones

A central question is whether clones do actually complicate maintenance and cause bugs. To answer these questions the

evolution of clones has been studied. The following provides a brief summary. An extensive overview is given by Pate and colleagues [7]. It was found that many clones are short-lived, while others that live longer rarely ever change [8], [9]. Comparisons of the change-frequency of cloned and non-cloned code—often referred to as clone stability—showed that, in many systems, cloned code changes even less frequently than non-cloned code [10], [11]. If clones change often they may be more difficult to maintain. Each change also bears the risk of unwanted inconsistencies. Some studies reported on cases where cloned code, indeed, changed more often [12], [13], but it could not be found that clones cause more changes in general.

Other studies investigated how often changes to clones are carried out inconsistently and how often such changes caused new bugs or led to the incomplete removal of existing ones. Inconsistent changes appear frequently [14]. In some systems they accord for more than half of all changes to clones [9], [15]. Cases in which such inconsistencies cause bugs exist and cannot be ignored [16]. A controlled experiment, in which developers were assigned bug-fixing tasks, indicated a high risk of incomplete bug-fixes when bugs are cloned [17]. Nevertheless, a general relation between clones and bugs could not be established [18], [19]. That is, some clones do cause bugs, but most do not.

B. Clones and Developers

The first to analyze code clones against the background of developers where Kim and colleagues [20]. They defined a taxonomy of the developer intent behind cloning after they observed programmers at work. They found that developers often recall the clones they have created and that this knowledge is difficult to transfer to other programmers. Nevertheless, they did not investigate how the interaction of developers affects clones. Balint and colleagues applied an automatic approach to detect who is changing the clones in three open-source software systems [1]. Using the `blame` functionality of `CVS`, they analyzed who made the last change to each line of a clone and visualized the data in a clone evolution view. They report on reoccurring patterns of changes to clones, including cases where developers change clones, which were created by others, inconsistently. In contrast to our study, they did not quantify their observations. Neither did they investigate whether the inconsistent changes were made intentionally or whether they caused bugs.

Cai and Kim correlated the lifetime of clone genealogies with the number of developers who make changes to the containing files [2]. They found that the more developers are involved, the longer a clone survives in the system. They conclude that clones maintained by multiple developers may be more difficult to remove because a longer life-time was observed for them. Nevertheless, the reason for this relation was not investigated and the findings are solely based on correlation analysis. Our study, in contrast, analyzes who *created* the cloned code, provides statistics how often clones are authored by different developers, how often inconsistent

changes occur in these cases, and whether problems, such as bugs, occur.

C. Code Authorship

The detection of the author of a code entity has been a concern to other studies previously. The approaches can roughly be partitioned into two categories. First, the contributions of different developers can be quantified on the file level. Cai and Kim’s work uses such an approach, but does not provide detail how the data are gathered [2]. Weyuker and colleagues investigated whether the number of developers who contributed to a file can be used to enhance bug prediction models [21]. They found that this metric does not play a decisive role in this context. In a follow-up study by Bird and colleagues the notion of authorship was refined with contribution networks. These are also based on file-level data [22]. Their results indicate that bugs are more likely to occur when developers change code they are not experienced with. An approach by Gîrba and colleagues extracts the number of changed lines from CVS log messages to calculate the code ownership on the file level. They introduce a time-line visualization that shows code ownership and how it changes over time [23]. These metrics are easy to compute from the metadata of a source code management system (SCM), but are not precise enough for a study on clones. Clones are local passages of source code which cannot be effectively analyzed on the file level.

The second category of approaches is the line-based authorship detection. The aforementioned study on clones by Balint and colleagues [1] uses CVS `blame` that provides the revision in which each line was modified last and the user who applied this change. Rahman and Devanbu use the same functionality of `git` to investigate who contributed to buggy fragments of code and conclude that the lack of experience in a particular file is a factor that may induce bugs [24]. Although such authorship detection methods provide data on a much more fine-grained level than file-based techniques, they are still not accurate enough for our study. Even if just one character is modified, the whole line will be regarded as changed, which leads to an overestimation of changes. Furthermore, code formatting—often carried out automatically and in the large—can heavily distort the authorship detection. More imprecision is caused by the fact that the tools used to obtain the author only provide who made the most recent change to a line. The previous history remains hidden.

In this paper we use an incremental approach that is different from both categories. It will be described in Section III.

D. Authorship vs. Ownership.

The question, who contributed a piece of code, is frequently discussed under the term of *code ownership*. Being the owner of a thing implies possession, control, and responsibility at a certain point of time. These properties cannot be extracted from source code and its evolution alone. For instance, some code may not change over some time, but the responsibility for

it may be transferred to another developer (e.g., when the previous owner leaves the project). SCM metadata only tells who added, modified, or deleted code. Hence, we speak of *code authorship*, since authoring does only imply the creational aspect. This term is more precise for what approaches such as the aforementioned and ours can provide. It also avoids confusion and misinterpretation of the results.

III. AUTHORSHIP DETECTION

As described in Section II, existing approaches for authorship detection are too inaccurate for our purpose. Consequently, we define a new approach that overcomes the shortcomings of the existing ones. The general idea is to incrementally track the author of each token in the source code over all revisions recorded in the code repository. Tokens are the smallest meaningful unit in the source code. Detecting authors on this level has the advantage that layout changes do not affect the measurements, because tokens abstract from whitespace and line-positions. By obtaining the authorship information from all available revisions, we further ensure that no information is lost. The authors of code fragments and clones are aggregated from the detected token-authors. Changes to the code are extracted from a Subversion (SVN) repository. SVN was chosen although newer systems such as `git` provide better metadata. We aimed to analyze long-lived systems that provide complete records of their history. Subversion has been widely used for more than a decade and many projects used it during their whole life-time. In the following our approach is explained in detail.

A. Token Authorship

The author of a token is the user who originally added it to the code. Whenever a developer adds a token it will be assigned to him. If existing tokens are changed, we decompose this action into a deletion and an addition of tokens. The newly added tokens will be assigned to the person who made the change. The change analysis differentiates between two kinds of changes: (1) changes inside a file, that is, changes to the file’s token stream and (2) changes on the file-system level, such as copying, moving, or deleting files. Changes to the token stream change the author information as described before, whereas changes on the file-system level are handled differently. When a file is moved from one location to another, the authorship of the tokens inside will not change. When a file is copied, the person making the copy will become the author of all tokens in the file. This distinction is made because we regard the copying of files as the creation of new clones, whereas moving files should not affect authorship. A file may undergo both types of changes (token and file-system level) in the same SVN commit. In such cases we split the change into two operations: (1) a token level operation and (2) a file-system level operation. As an example, if person p creates file f and person p' moves that file to another location f' , all tokens in f' will still be authored by p . When p' moves and changes f in the same revision, only the newly added tokens in f' will be assigned to p' .

Formally, we analyze all relevant revisions $r \in R$ from a project's SVN repository. A revision is relevant, if it contains at least one change to a file of the analyzed programming language. A revision is a tuple $r = (F, C, a)$, consisting of a set of existing files F , a set of changes to files C as they appear in SVN's revision log, and an author a who is the committer of r according to the SVN log (which we also will refer to as $a = \text{author}(r)$). For each file we can extract its list of tokens in a specific revision as $T(f, r)$. For each token $t \in T(f, r)$ we determine its author a . This gives us a list of authors for all the tokens in the file $A(f, r)$ that is isomorphic to $T(f, r)$. That is, the author of the n -th token $t_n \in T(f, r)$ is the n -th author $a_n \in A(f, r)$. The calculated list $A(f, r)$ is stored along with the revision r so that it can be used for further analysis.

To obtain the token authors, we process all revisions $r_i \in R$ in the order in which they were added to the repository. For each r_i we process all changes $c \in C$ using the following rules:

1) *Additions*: If c is an addition of a new file that was not present in r_{i-1} , we create a new author list $A(f, r_i)$ with the same size as the token list $T(f, r_i)$ and assign the revision's author as the author of all tokens in the file f . Formally, $\forall a \in A(f, r_i) : a = \text{author}(r_i)$.

2) *Modifications*: If c is a modification of a file f that existed in r_{i-1} , we compute the differences between $T(f, r_{i-1})$ and $T(f, r_i)$ using the Longest Common Subsequence algorithm (LCS) on the tokens. This gives us a list of change deltas D between the two token lists. Each delta $d \in D$ is a tuple $d = (s, l, t)$, where s is the start index of the change, l is the length of the change in tokens, and t is the type of the change, which is either an addition or a deletion of tokens. The deltas are ordered as they appear in f from its beginning to its end. To apply the changes to f 's author information we first copy the previous author sequence to the current revision by assigning $A(f, r_i) = A(f, r_{i-1})$. Then we iterate over each $d \in D$ in reverse order and apply each d to $A(f, r_i)$ using the following rules: If d is a deletion, we remove l authors from $A(f, r_i)$ starting from s . If d is an addition, we insert $\text{author}(r_i)$ at position s into $A(f, r_i)$ for l times. This updates the author information for f in r_i . Afterwards $A(f, r_i)$ is isomorphic to $T(f, r_i)$. We process the deltas in D in reverse order, because thereby modifications are applied to $A(f, r_i)$ back-to-forth, which ensures that the starting positions of the deltas are not invalidated by previously processed modifications. If a deletion and an addition appear at the same token position (tokens have been replaced), the deletion is processed prior to the addition.

3) *Deletions*: If c is a deletion of a file f that was present in r_{i-1} , its author list $A(f, r_{i-1})$ will not be resumed in r_i .

4) *File Copies*: If c copies a file f that existed in r_{i-1} to a file f' , then f' will be handled as an added file according to the rules above.

5) *Unchanged Files and File Moves*: If a file f has no corresponding change c in r_i , or if it was just moved to f' in the file system, we copy the author information without changing it. That is, for unchanged files $A(f, r_i) = A(f, r_{i-1})$

and for moved files $A(f', r_i) = A(f, r_{i-1})$.

B. Code Authorship

The author information for tokens must now be aggregated for code fragments, which are sequences of tokens. A code fragment $frag = (f, r, s, l)$ is a continuous subsequence of the tokens from $T(f, r)$, where f is the containing file, r the revision, s is the start index in $T(f, r)$, and l is the length of the sequence. Since $T(f, r)$ and $A(f, r)$ are isomorphic, we can obtain the list of token authors of the fragment from $A(f, r)$ using s and l . The list $A(f, r)$ contains the author for each token in $frag$. We quantify the number of tokens from each author of $frag$. The *main author* of $frag$ is defined as the author who contributed most of the tokens. If two or more authors contributed the same amount of tokens to $frag$ one of them is randomly chosen as the main author. This may appear to be a too strong simplification of our accurate authorship tracking. Nevertheless, it is required to categorize clones according the number of authors. In the Section V we provide data how much of a fragment is authored by its main author on average. The vast majority of all fragments is predominantly authored by just one person. Even though we loose some accuracy by defining a main author we still benefit from the fact that the token-based approach is not extensively distorted by formatting. Comparisons with SVN's `annotate` command have shown that these would strongly affect the results in some systems.

C. Clone Authorship

Our definition for code authorship, finally, needs to be transferred to code clones. In our study clones are exact copies of code fragments. Such a fragment may be copied one or more times. All equal fragments together form a *clone class*. Defining the author of clone fragments is straightforward because they are code fragments for which we defined authorship before. Hence, we can obtain the main author for each clone fragment. In our study we are interested in the distinction between clone classes that have just one author and such that have multiple authors. We now can separate the set of all clone classes CC into such classes whose fragments have different main authors as CC_{multi} and such classes whose fragments have the same main author as CC_{single} . That is, CC_{single} are the single-author clone classes, whereas the multi-author clone classes are in CC_{multi} .

D. Sensitivity to initial checkins

Because of its incremental nature, our approach must not only visit every revision to provide accurate results, it also must begin with the very first revision of the project's history. If the analysis starts with a revision to which multiple developers contributed, it cannot decide who authored the tokens. Typically, SVN repositories begin with an import of some existing code. To avoid wrong author assignments in such cases, we mark all tokens that were added in the very first revision as unresolved. Such tokens will receive special handling. Clones that contain unresolved tokens over a certain

threshold will be excluded from the analysis. We chose subject systems that have only small initial checkins to limit the amount of unresolved clones. How many clones have been excluded per system will be reported in Section V.

E. Branches

Branches in SVN are effectively clones of the whole code base in the trunk. Nevertheless, they cannot be compared to clones created by copy-and-paste programming, because they are explicitly managed via SVN. Consequently, our approach analyzes only the trunk (the main branch) of each system. Another issue regarding branches is the possibility that new code is created by different developers in a branch and then merged into the trunk by one developer. Although SVN documents branch creation in its metadata, earlier versions did not document merge operations between branches. Changes incoming from other branches cannot be reconstructed from the data. In such cases the changes will be assigned to the developer who makes the merge. To reduce the effect of such cases on our measurements, we choose subject systems where the development mainly happens in the trunk and branches are mainly used to propagate bug-fixes to previous releases. We will further discuss this in Section IV.

IV. STUDY PROCEDURE

We use the approach described in section III to detect the authors in five different software systems. In the following we describe how we detected clones and which systems have been analyzed.

A. Clone Detection

The incremental clone detector `iclones`¹ was used to detect clones in all relevant revisions. `iclones` detects clones in every revision using a token-based detection approach based on a generalized suffix tree. When it proceeds to the next revision it keeps its internal data structures and updates the information for files that have been added, changed, or removed. Göde and Koschke give a detailed explanation of this approach in [25]. `iclones` does not only detect clones in consecutive versions, it also tracks clone fragments across versions using a token-based LCS approach that is similar to the one we use for authorship detection. Based on this difference information it is determined whether the fragments of a clone class have been changed from one revision to the next. To this end, `iclones` tracks the fragment positions across all revisions and checks for each revision r whether the fragment f 's token stream has changed by comparing $T(f, r_i)$ and $T(f, r_{i+1})$ using LCS. If at least one fragment of a clone class has changes, the clone class will be marked as changed. The creation of a new fragment will not be regarded as change in our case, because it is not a modification of cloned code.

Furthermore, `iclones` detects whether the fragments of a clone class have been changed consistently. A clone class is changed consistently, when the change deltas applied to all its fragments according to LCS are identical. If at least

one fragment has a different set of change deltas, the change is inconsistent. The detection of consistent and inconsistent changes is further explained in [9].

The results of a clone detector strongly depend on its configuration. In the following we describe how `iclones` was configured and which precautions were taken to improve the quality of the results.

1) *Clone Types*: We decided to detect only exact clones (in contrast to near-miss clones where the fragments may differ to a certain extent), because we differentiate between consistent and inconsistent changes to clones. While change consistency can be exactly defined for exact clones it is more difficult to define for near-miss clones, in which the fragments have differences. An inconsistent change to a near-miss clone may make its fragments actually more similar by removing differences. This is not the kind of inconsistency we are interested in.

2) *Minimum clone length*: For the minimum length of a clone we chose 50, 100, and 150 tokens. Because the results regarding authorship did not vary much between these three data sets, we will only report results for the 100 token setting in the remainder of this paper. In practice 100 tokens roughly correspond to 10 to 15 lines of code.

3) *Code Exclusion*: We excluded code that usually contains coincidental clones, which do not necessarily stem from copy and paste. To this end, files that define test cases or contain generated code have been excluded. A similar case are lists of import statements and long array initialization sequences. We removed these artifacts from the token stream automatically prior to clone and authorship detection.

4) *Method separation*: Token-based approaches may detect many clones that only consist of the end of one method and the signature of the next. Such artifacts are not caused by copy and paste. To avoid them, we detect method boundaries in the token-stream using pattern matching and insert artificial delimiter tokens between methods. The clone detector will stop detecting a clone when it encounters such a delimiter. That is, clones that span over multiple methods are separated at the method boundaries. The resulting clones are kept, if they exceed the minimum clone length.

A clone detection approach based on abstract syntax trees (AST) would have provided the syntactical information, too. Nevertheless, such approaches are difficult to use in evolutionary studies on long-lived systems as ours. It cannot be assumed, that the code in each and every revision will be syntactically correct, which is a requirement for these techniques. Furthermore, AST-based approaches demand more time and space for their computations, which is critical when, as in our study, more than 10,000 revisions are analyzed.

B. Subject Systems

We analyzed the history of five software systems from different domains. Details on systems are listed in Table I. Only relevant revisions are reported in the revision column. The column labeled *max. coverage* reports the maximum clone rate, that is, the highest percentage of cloned tokens among all

¹<http://www.softwareclones.org/iclones.php>

TABLE I
SUBJECT SYSTEM PROPERTIES

System	Language	Revisions	Begin	End	Years	max. KLOC	max. coverage [%]	Committers
<i>Ant</i>	Java	8,774	2000-01-13	2011-08-19	11.6	215.0	2.1	46
<i>FindBugs</i>	Java	10,339	2003-03-24	2012-07-26	9.4	264.3	14.9	28
<i>FreeCol</i>	Java	10,930	2005-11-02	2013-01-30	7.3	184.8	4.0	35
<i>Handbrake</i>	C	1,970	2006-01-14	2013-04-01	7.2	88.4	4.8	25
<i>swclones</i>	Java	1,378	2009-06-16	2012-03-21	2.8	58.2	5.0	18

tokens, the systems reached during their lifetime. The number of committers is the number of unique users who committed at least one relevant version. Possible alias names, which are used by some users, have been merged as far as they could be reconstructed.

All five systems either did not use branches at all or used them mainly to maintain old releases. If branches were used for other purposes we will describe these in the following.

1) *Apache Ant*: *Ant*² is an open-source build-automation system for Java development. For our study we analyzed its `core` component. Several proposals for a next-generation Ant have been made and developed in the *proposals* folder of the code repository, which can be regarded as a branch folder. Some proposals started with a clone of the code base that was heavily refactored and changed. Nevertheless, all proposals have been discarded eventually. Consequently, they have been excluded.

2) *FindBugs*: *FindBugs*³ is an open-source static analysis tool for Java source code. It searches for typical patterns that indicate potential programming errors. The high maximum clone coverage is caused by an outlier revision. For most of the time it lay between 0.5% and 3.0%.

3) *FreeCol*: *FreeCol*⁴ is an open-source reimplement of the commercial strategy game *Colonization*. It has been newly developed from scratch by a team of enthusiasts and is written in Java. It reaches its maximum clone coverage in the middle of the analyzed period. From there on, the coverage steadily declined to 0.7% in the last revision.

4) *Handbrake*: *Handbrake*⁵ is an open-source multi-platform video transcoder written in C. It consists of a central library that performs the transcoding and three GUI applications for Linux, Mac and Windows. The Windows GUI was left out in this study, because it is written in C# and not in C.

5) *softwareclones.org*: Our research group develops tools for clone detection and analysis in Java⁶. These are the tools that have been used to execute this study. For brevity we will refer to them as *swclones* in the remainder of this paper. The whole code repository including tools for clone detection, analysis, visualization, and transformation was analyzed for this study. The code is developed by a team of 18 researchers and student assistants. Although the main focus of the tools are

²<http://ant.apache.org>

³<http://findbugs.sourceforge.net>

⁴<http://www.freecol.org>

⁵<http://www.handbrake.fr>

⁶<http://www.softwareclones.org>

TABLE II
FRAGMENT MAIN AUTHOR SHARES AND EXCLUDED CLONE CLASSES

System	All [%]	CC_{single} [%]	Excl. [%]
<i>Ant</i>	90.7	95.0	3.245
<i>FindBugs</i>	98.7	99.0	0.000
<i>FreeCol</i>	80.1	94.9	13.767
<i>Handbrake</i>	94.9	98.1	0.004
<i>swclones</i>	96.5	96.0	0.000

code clones, clone detection has only been used sporadically during the development process.

V. RESULTS

Before we pursue our research questions, we take a look on the feasibility of our approach for the chosen subject systems. The idea of the main author is based on the assumption that code fragments are usually predominantly authored by one author. If this is not the case and different authors equally contribute to the fragments, our classification would be inaccurate. Consequently, we measure the average percentage of tokens the detected main author owns in a clone fragment. The results are shown in Table II. We present the averages for all fragments, and the value for the fragments of only CC_{single} classes. The second value is important to validate whether CC_{single} can indeed be regarded as being mainly authored by just one person. Most of the values are roughly 95% or higher. For *Ant* and *FreeCol* the values for all fragments are lower. However, it can still be claimed, that the detected main author made the essential contributions, since no value lies below 80%.

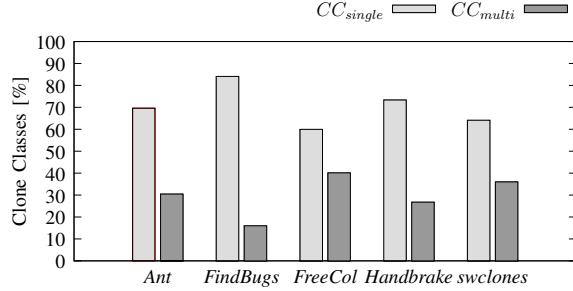
For three of the systems, the SVN history started with the import of an existing code base. That is, for these systems all tokens of the initial revision have been marked as unresolved. As soon as more than one percent of a fragment's tokens are marked as unknown, we excluded the fragment and its whole clone class from the analysis. Higher values of this threshold (up to 5%) did not change the exclusion rate significantly. The exclusion of clone classes is only notable for *Ant* and *FreeCol*. For the latter, 13.8% had to be excluded, because the first 600 revisions in its repository seem to be corrupted and were skipped.

A. Clone authors

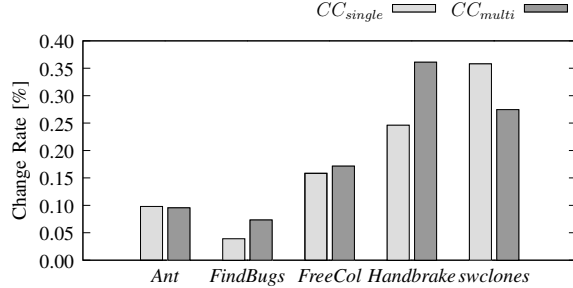
To answer Question 1, that is the question how often clones are authored by more than one developer, we first detect clones

TABLE III
FREQUENCY OF DIFFERENT CLONE CLASS TYPES (IN THOUSAND)

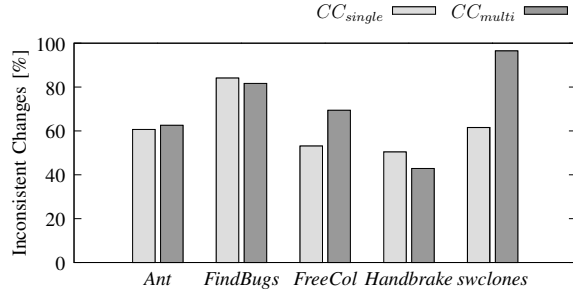
System	All	CC_{single}	%	CC_{multi}	%
<i>Ant</i>	138.1	96.1	69.6	42.0	30.4
<i>FindBugs</i>	421.0	353.9	84.1	67.1	15.9
<i>FreeCol</i>	413.3	247.6	59.9	165.8	40.1
<i>Handbrake</i>	72.7	53.3	73.3	19.4	26.7
<i>swclones</i>	28.4	18.2	64.1	10.2	35.9



(a) Relative frequency.



(b) Change rate.



(c) Rate of inconsistent changes.

Fig. 1. Comparison of CC_{single} and CC_{multi} clone classes

in every relevant revision of each system. All detected clones are categorized into CC_{single} and CC_{multi} and counted. If a clone class appears in more than one revision, each occurrence will be counted. The decision whether a clone class is CC_{single} or CC_{multi} is made for each occurrence. Table III shows the results which are also visualized in Figure 1a.

For all systems, both CC_{single} and CC_{multi} do exist in a notable quantity. The number of CC_{single} is always higher than the number of CC_{multi} clone classes. That is, most of the clones we have found in the project histories are mainly

TABLE IV
CHANGES TO CLONES

System	Sample	CR [%]	CC [%]	IC [%]
<i>Ant</i>	CC_{single}	0.098	39.4	60.6
	CC_{multi}	0.095	37.5	62.5
	all	0.097	38.8	61.2
<i>FindBugs</i>	CC_{single}	0.039	15.9	84.1
	CC_{multi}	0.073	18.4	81.6
	all	0.044	16.6	83.4
<i>FreeCol</i>	CC_{single}	0.158	46.9	53.1
	CC_{multi}	0.171	30.6	69.4
	all	0.164	40.1	59.9
<i>Handbrake</i>	CC_{single}	0.246	49.6	50.4
	CC_{multi}	0.361	57.1	42.9
	all	0.277	52.2	47.8
<i>swclones</i>	CC_{single}	0.358	38.5	61.5
	CC_{multi}	0.274	3.6	96.4
	all	0.328	28.0	72.0

authored by just a single developer. Nevertheless, 15 to 40% of the clone classes contain fragments that are mainly authored by different developers.

B. Change frequency

To answer Question 2 we analyze whether CC_{single} and CC_{multi} change differently. First, we analyze whether clone classes of the two categories have the same probability to change. For each clone class we detect in revision r_i , we check whether its fragments are changed in the transition to r_{i+1} . If the tokens of at least one fragment were changed, the clone class is marked as changed. The change rate of CC_{single} clone classes can then be computed as the fraction of the changed CC_{single} clone classes and the total number of CC_{single} clone classes. The change rate of CC_{multi} clone classes is computed accordingly. That is, these change rates reflect the probability that a clone class, found in one revision, is changed in the next revision.

Table IV shows the results. The CR column represents the change rate of the respective clone classes. Figure 1a illustrates the change frequency of CC_{single} and CC_{multi} for each system in comparison. First of all, it is to be noted that clones do not change too often in general. In all systems less than 0.4% of all clone classes, we detected during the system's evolution, changed in the next version. As an example, we detected 138,096 clone classes in all revisions of *Ant*, but only 134 of these changed within the 11.6 years we analyzed. These data support findings of a previous study on this topic [9]. The clones in *Handbrake* and *swclones* have a notably higher change rate compared with the clones in the other systems. Both systems have a much shorter history in terms of revisions than the others. It is also these two systems which change rates for CC_{single} and CC_{multi} expose the largest difference. In *Handbrake* the change probability of CC_{multi} clone classes is 55.5% higher than the change probability of CC_{single} clone classes. In *swclones* the opposite is the case and CC_{single}

clone classes change more frequently. The highest relative difference can be observed in *FindBugs* where the change rate of CC_{multi} is twice as high as for CC_{single} . For *Ant* and *FreeCol* no significant difference was measured.

In summary, there is no systematic difference that indicates a general effect clone authorship has on the probability that a clone class changes. The effect seems to be system dependent.

C. Change consistency

We further analyze the consistency of the changes to answer Question 3. If a clone class changed and all its fragments underwent identical changes, we consider the change to be consistent. If the fragments changed differently or some fragments did not change, while others in the same class did, we consider the change as inconsistent. Table IV shows how often CC_{single} and CC_{multi} changed consistently or inconsistently. Figure 1c shows only the percentage of inconsistent changes in comparison for the subject systems. For most systems the rate of inconsistent changes does not differ notably for CC_{single} and CC_{multi} . In *FreeCol* and *swclones* CC_{multi} are changed more often inconsistently than CC_{single} . In *Handbrake* CC_{single} have a slightly higher rate of inconsistent changes. Again, there is no systematic difference in the analyzed systems.

D. Intentions and Bugs

To answer question 4 and to learn more about the intentions and possible defects connected to CC_{single} and CC_{multi} clone classes, we manually inspected a large part of the results. To this end, the clone evolution visualization tool *cyclone*⁷ was used [26]. *Cyclone* visualizes clone histories, which eases the reconstruction of changes and intentions. For *swclones* and *Ant* we evaluated each change to clones on four levels: (1) For inconsistent changes, whether the inconsistency was intentional, (2) whether the inconsistency caused a bug or is an incomplete bug removal, (3) the rationale of the cloning and the changes to the clones, (4) whether the fragments of the changed clone class were located in the same or different files.

1) *swclones*: We inspected all changes to the clones in *swclones*, because for this system we measured the largest difference in change consistency. Multi-author clones changed more often inconsistently than single-author clones. Furthermore, we have in-depth knowledge about the code which allows us to judge the intentions behind changes and whether an inconsistency is a bug.

The inconsistent changes to clones were mostly intentional for both single-author and multi-author clones. Among the multi-author clones only one out of 25 inconsistent changes was unintentionally inconsistent. A bug-fix regarding the canonicalization of path names was applied only in the default input module of our clone detector. A similar module for network input contains the same bug, which was not fixed. Nevertheless, this module has not been used since. Hence, we

regard this case as rather unproblematic. Surprisingly, 3 out of the 38 inconsistent changes to single-author clones (7.9%) were unintentional and caused actual bugs that still existed in the most recent revision. They have in common that the bugs are located in complex algorithms and that their cloned occurrences lie closely together in the same file. In each case only one of two bug occurrences was fixed.

Regarding bugs we found that multi-author clones were less often subject to bug-fixing activity compared with single-author clones (7.7% vs. 20.6%). Most of these fixes, except the ones mentioned above, were applied consistently.

Single-author and multi-author clones differ in their rationale. Almost all multi-author clones (21 out of 27) are copies between two different viewer applications. One of these was started by copying existing GUI components from the other. The plan to reintegrate both applications at a later point of time was soon abandoned, because it was found that the different functional requirements could be better satisfied in separate tools. Consequently, the two GUIs purposefully developed independently which caused the high volume of inconsistent changes to multi-author clones in *swclones*. The remaining changes to multi-author clones were related to similar cloning across different implementations of the same architectural component.

Not one multi-author clone class had all its fragments in the same file. In contrast, the single-author clones lie to 57.7% in the same file. They often copy logic snippets that are difficult or impossible to unify. For instance, because they implement the same logic for different primitive types.

To summarize, multi-author clones were mostly caused by file cloning, which was less often the case for single-author clones. Furthermore, the changes to multi-author clones contained less problematic cases, such as bug fixes, compared with single-authored ones.

2) *Ant*: As second system, we inspected all 134 changes to clones in *Ant*. According to our statistics single-author and multi-author clones do not differ significantly regarding their change frequency and their change consistency. Since we are not familiar with the code, we cannot safely judge whether inconsistent changes were made intentionally. Nevertheless, we can analyze how single-author and multi-author clones differ in this system. We further inspected the commit messages to evaluate whether the changes were bug fixes.

We found bug-fixing activity to be more common in single-author than in multi-author clones (31.2% vs. 13.2%). Inconsistencies that were obviously unintended, because the missing changes were propagated to all fragments later, rather occurred in the single-authored clones. We clearly identified three such cases for single-author clones, one of which took three attempts over eight months to be fixed consistently. Among the changes to multi-author clones only one such late propagation was found. In this case the inconsistency was fixed in the following revision where the clone was also removed.

The rationale of cloning is less discriminative than in *swclones*, the tendency, however, is the same. The changed multi-author clones lie to 25.64% within the same file. For

⁷<http://www.softwareclones.org/cyclone.php>

single-author clones this is the case in 30.11%. This smaller difference may be caused by *Ant*'s highly generic architecture that decomposes the build-process into tasks, each of which are implemented in their own class. These tasks are often similar, especially when they implement similar functionality, such as different commands of the same source code management system.

More interesting is why the clones are changed. Most of the consistent changes to multi-author clones are directed to warnings from static defect checkers, which are usually issues of style, performance, program comprehensibility, or typical programming mistakes. These issues are automatically detected by tools. We did not consider them as bugs. If they are cloned, the tool will point out all locations to the user automatically. Hence, it does not come as a surprise that these are issues are removed consistently. While 71.4% of the consistent changes to multi-author clones are directed to such issues, only 35.1% of the consistent changes to single-author clones fall into this category. Among the inconsistent changes, such issues are rarely the reason for the change (9.1% for multi-author clones vs. 5.6% for single-author clones).

To summarize, we did not find multi-author clones to be related to problematic changes more often than single-author clones. Most inconsistencies are intentional and the rare cases of clearly unintentional inconsistencies were rather found among the single-author clone classes. When multi-author clone classes changed consistently, the reason mostly were automatically detected warnings about coding conventions and style.

VI. DISCUSSION

Our analysis of code authorship was motivated by the conjecture that problems with clones may be caused when the copied fragments were authored by more than one developer. If this was the case, the authorship of clones could be used to prioritize clones in clone management and to develop measures to prevent such problems. Our results show that multi-author cloning is common in software, albeit most clones are created and mainly maintained by just one developer. The analysis of the change frequency and consistency did not reveal a notable and systematic difference for the multi-author clones. The analysis of further systems may reveal a general tendency towards single-author or multi-author clones. Nevertheless, to be a useful metric for clone management, multiple authors would need to have an effect of considerable size on the harmfulness of clones, which they do not seem to have. Given the small number of changes to clones in general, the relative differences we measured for the change rate and the consistency translate to a very small absolute number of actual cases. To be practically relevant the differences had to be much larger.

Our manual inspection further contradicts our initial conjecture, as the few harmful clones we found were rather single-author than multi-author. The difference between these two kinds of clones is certainly not significant. Nevertheless, our analysis indicates the possibility that single-author clones

could be more likely to cause harm. A possible explanation lies in the different rationales we observed for single-author and multi-author clones. As our manual inspection revealed, multi-author clones tend to be larger copies of whole structures, whereas single-author clones were more often smaller copies of complex algorithmic parts. As an example, an inconsistency in the layout of two GUI dialogs should be less harmful than an inconsistency in some critical part of the application logic. We observed this tendency for both *swclones* and *Ant*.

In both systems, for which we did manual inspections, we observed that consistent changes were very often directed to style issues, while inconsistent ones rarely are. When multiple authors were involved in the cloning, the amount of style related changes was higher in *Ant*. This is also true for *swclones*, but this result is based on only one existing consistent change to multi-author clones.

Our study investigates the effect of multiple developers of clones from only one perspective. Besides the question whether it makes a difference if different developers create and maintain the copied fragments, it may also be asked, whether unwanted inconsistencies occur when a clone authored by one developer is changed by another. This, however, is another question that should be asked in broader context because changing someone else's code may cause problems in general and may not be limited to clones only. Future research should be directed to this question.

VII. THREATS TO VALIDITY

In the following we describe possible threats to the validity of our study.

A. Internal Validity

The results of our study depend on the metadata provided by *SVN*. As any other SCM it can only provide what was explicitly inserted. Information may be lost if files are moved or copied without using *SVN*'s commands for this purpose. In such cases, the authorship may change incorrectly. If changes are proposed as patches by an external developer, the project member who commits the changes appears as their author. This is not exactly true, however, the committing author is still responsible for the correctness of the changes she or he makes. In some projects the same developer contributes under different user names. We merged such users as far as we could reconstruct their aliases.

Another possible threat is the Longest Common Subsequence (LCS) algorithm, which we use to analyze changes to the token stream. In practice, LCS is usually able to reconstruct the changes that were applied to a sequence correctly. Nevertheless, is not able to track code movements inside a file. If, for instance, a method is moved inside a file, our approach will regard this as a deletion and an addition that was authored by the developer who moved the method. Such cases may appear in refactorings. To evaluate whether such changes do largely affect the authorship detection, we analyzed how the amount of authored tokens changes over time for each developer. We did not find abrupt changes in authorship, which would be

visible if such situations occurred frequently and affected the measurements thoroughly.

B. External Validity

Although we chose multiple long-lived systems from different domains, we cannot generalize our results on all other software systems. Most of the analyzed systems are written in Java and developed as open-source software. Systems written in other languages or in an industrial context may expose different properties. Furthermore, our results indicate that the relationship between developers and the effects of clones, may be system dependent. Future research should analyze further systems, written in different languages and with an industrial context.

The results of our study are valid for exact clones. We did not analyze clones with different variable names or gapped clones, because the definition of change consistency is problematic for these. Inconsistent changes may split previously found exact clones into multiple classes. In our manual inspection of all changes to the clones in *swclones* and *Ant*, in which we also analyzed clone genealogies, such cases did not appear frequently. Future research should further investigate how multiple authors affect the evolution of near-miss clones.

VIII. CONCLUSION

In this study we analyzed how multiple developers affect the evolution of exact code clones. Indeed, we found differences between single-author and multi-author clones. Nevertheless, our results do not indicate that the involvement of more than one developer is a typical cause of unwanted inconsistencies and clone-related bugs. The differences we have found are rather directed to the rationale of cloning and the changes applied to the clones. The multi-author clones in the analyzed systems seem to have an even reduced risk potential, because the inconsistent development is intended or changes are mostly directed to style issues and do not affect semantics. These first results, however, certainly require to be further confirmed by future research.

Because of the absence of a strong relation between the number of authors of a clone and the probability of unintended inconsistent changes and bugs, we cannot recommend to use the number of authors of a clone as a metric to prioritize clone management activity.

ACKNOWLEDGMENTS

We would like to thank Marcel Steinbeck for his support on implementing technical features of our analysis. This work was supported by a research grant of the *Deutsche Forschungsgemeinschaft* (DFG).

REFERENCES

- [1] M. Balint, R. Marinescu, and T. Girba, "How Developers Copy," in *International Conference on Program Comprehension*, Jun. 2006, pp. 56–68.
- [2] D. Cai and M. Kim, "An empirical study of long-lived code clones," in *International Conference on Fundamental Approaches to Software Engineering*, 2011, pp. 432–446.
- [3] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *International Conference on Software Engineering*, 2012, pp. 255–265.
- [4] J. Harder and N. Göde, "Quo vadis, clone management?" in *International Workshop on Software Clones*, 2010, pp. 85–86.
- [5] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, no. 06301, 2007.
- [6] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.
- [7] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone evolution: a systematic review," *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 261–283, 2013.
- [8] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, 2005, pp. 187–196.
- [9] N. Göde, "Evolution of type-1 clones," in *International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 77–86.
- [10] J. Krinke, "Is cloned code more stable than non-cloned code?" in *International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 57–66.
- [11] J. Harder and N. Göde, "Cloned code: stable code," *Journal of Software: Evolution and Process*, 2012, published online.
- [12] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software," in *Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, 2010, pp. 73–82.
- [13] M. Mondal, C. K. Roy, and K. A. Schneider, "An empirical study on clone stability," *SIGAPP Applied Computing Review*, vol. 12, no. 3, pp. 20–36, Sep. 2012.
- [14] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *Working Conference on Reverse Engineering*, 2007, pp. 170–178.
- [15] S. Bazrafshan, "Evolution of near-miss clones," in *Conference on Source Code Analysis and Manipulation*, 2012, pp. 74–83.
- [16] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *International Conference on Software Engineering*, 2009, pp. 485–495.
- [17] J. Harder and R. Tiarks, "A controlled experiment on software clones," in *International Conference on Program Comprehension*, 2012, pp. 219–228.
- [18] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *Working Conference on Mining Software Repositories*, 2010, pp. 72–81.
- [19] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *International Conference on Software Engineering*, 2011, pp. 311–320.
- [20] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *International Symposium on Empirical Software Engineering*, 2004, pp. 83–92.
- [21] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 539–559, Oct. 2008.
- [22] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't Touch My Code! Examining the Effects of Ownership on Software Quality," in *European Software Engineering Conference and Symposium on The Foundations of Software Engineering*, 2011, pp. 4–14.
- [23] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," in *International Workshop on Principles of Software Evolution*, 2005, pp. 113–122.
- [24] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *International Conference on Software Engineering*, 2011, pp. 491–500.
- [25] N. Göde and R. Koschke, "Incremental clone detection," in *European Conference on Software Maintenance and Reengineering*, 2009, pp. 219–228.
- [26] J. Harder and N. Göde, "Efficiently handling clone data: RCF and Cyclone," in *International Workshop on Software Clones*, 2011, pp. 81–82.