An Empirical Study of Clone Removals

Saman Bazrafshan, Rainer Koschke
Software Engineering Group
University of Bremen, Germany
{samran.bazrafshan,koschke}@informatik.uni-bremen.de

Abstract—It is often claimed that duplicated source code is a threat to the maintainability of a software system and that developers should manage code duplication. A previous study analyzed the evolution of four software systems and found a remarkable discrepancy between code clones detected by a state-of-the-art clone detector and those deliberately removed by developers as the scope of the clones hardly ever matched. However, the results are based on a relatively small amount of data and need to be validated by a more extensive analysis. In this paper, we present an extension of this study by analyzing deliberate as well as accidental removals of code duplication in the evolution of eleven systems. Based on our findings, we could confirm the results of the previous study. Beyond that we found that accidental removals of cloned code occur slightly more often than deliberate removals and that many clone removals were in fact incomplete.

Index Terms—Clone removal, clone evolution, software maintenance

I. INTRODUCTION

It is often claimed that duplicated source code—so called clones—increases the maintenance effort in software systems and, therefore, is harmful. This claim is based on the assumption that because of clones the source code becomes redundant, change effort increases, comprehensibility decreases, and inconsistent changes can introduce new defects or prevent the removal of existing ones. Nonetheless, recent studies [12], [19], [21], [27], [28] showed that clones cannot be considered a threat to software maintenance in general. Copy&Paste programming is a common practice to speed up software development, to reuse existing and reliable code, or to use existing code as starting point for new functionality [20], [25].

To support developers to keep track of and handle clones of clone management tools have been introduced that detect clones [22], [30], support refactorings [4], [16] and change propagation [9], [34] as well as monitoring to prevent unwanted inconsistencies [29]. The efficiency of such tools has been successively improved over the years so that even very large systems can be efficiently analyzed. Still, clone management has not yet become an integral part of the daily work of programmers. A major hindrance to the use of clone management tools is a missing relevance ranking of clones. Because results provided by state-of-the-art clone tools are based on only structural similarity in the source code, users are exposed to a vast number of clone information. The amount of data increases the effort or makes it impractical to filter useful clones just manually. Therefore, clones that have been automatically detected need to be ranked by the relevance to a specific maintenance tasks. Studying clones that were actually removed by developers may give indicators for a suitable relevance ranking.

For this reason, Göde [11] investigated deliberate clone removals in the evolution of four open-source systems to campaign for more observance of a maintainer’s view. Göde found a large discrepancy between clones detected by a state-of-the-art clone detector and code clones removed by developers as the scopes of the clones hardly ever matched. His results suggest that value can be added to clone management tools by considering knowledge about duplicated code that has been selected for refactoring by a programmer.

In this paper, we extend the previous study as it is based on a relatively small number of subject systems and less than one year of the system’s evolution has been analyzed. To validate the results in a more extensive analysis, we analyze the evolution of eleven open-source systems over a period of two years and investigate further characteristics of clone removals or refactorings, respectively. Based on the findings of this study, we will answer the following research questions:

Question 1 — How often and by what kind of refactorings is duplicated code deliberately removed?

Just as Göde [11] did, we investigated whether and with what frequency code clones have been deliberately removed by developers. An answer gives a clue about the programmer’s view on managing duplications. In addition, the refactorings applied to remove duplicated code are compared to clones detected by a state-of-the-art detection tool. As a result of the comparison existing weaknesses in automated detection can be uncovered and eliminated.

Question 2 — How often and by what kind of refactorings is duplicated code accidentally removed?

Apart from deliberate removals that have been subject of the previous study by Göde, clones might be removed accidently as a side effect by some other refactorings. We analyzed how often duplicated source code is removed by arbitrary code modifications by chance. Frequent accidental clone removals could be a reason for developers not to manage clones actively as the problem will resolve itself in the long-term. Moreover, information on refactorings that were actually not meant to remove cloned code but even so did, might contribute to the improvement of clone management tools by uncovering characteristics that indicate good clone candidates for a removal—focusing exclusively on deliberately removed clones might not reveal these characteristics.
Question 3 — Are there clones that are missed by a refactoring?

Besides analyzing clones that have been removed by refactorings, we also investigate whether or not duplicated code that could have been removed in consequence of a performed refactoring were missed by developers—which was no subject of Göde’s study. Developers may miss clones when performing a suitable refactoring if they are not aware of them. This risk could be mitigated using automated tool support that provides useful information.

Question 4 — What kind of measurable code characteristics may help in ranking clone candidates for removal?

Having detected deliberate and accidental clone removals in the evolution of software systems, it would be useful to extract code characteristics that flag clones to be removed. The approach presented by Göde [11] to detect clone removals, which has been adapted in this paper, is only semi-automated. Being able to use code characteristics to further automate the detection process would contribute to the improvement of clone management tools as the information helps to provide only meaningful data to the user. Göde investigated a few preliminary metrics, but his results suggest that those metrics do not clearly indicate good candidates. We will extend his preliminary study by analyzing additional metrics.

Contribution. To provide further insights and answer our research questions, we replicate and extend the study by Göde [11] who considered only a small number of subject systems and investigated exclusively those clones that were deliberately removed. To overcome these shortcomings in our study, we analyze the evolution of eleven subject systems over a two year time period. Moreover, we extend the study by also considering accidental clone removals and collect additional metrics that might indicate good candidates for clone removal through refactoring.

Outline. The remainder of this paper is organized as follows. Section II presents related work, including Göde’s study on deliberate clone removals. Our approach of analyzing clone removals is described in Section III. Section IV presents our case study and the results. Section V concludes.

II. RELATED WORK

Extracting and analyzing the evolution of clones have been subject to recent studies in clone research. This section summarizes previous work that is related to ours.

A. Clone Evolution

Antoniol and colleagues conducted a study on the extent and the evolution of clones in 19 releases of the Linux kernel [1]. They found that the amount of clones was rather small and that the clone ratio tends to remain stable across versions.

Kim and colleagues presented the first clone evolution model based on the mapping of code clones across multiple versions [21]. They investigated the evolution of clones in two Java systems by mapping clone classes1 of consecutive versions. Based on the results they concluded that the detected clones were either very volatile or hard to remove.

The evolution of the Linux kernel has also been studied by Livieri and colleagues who used metrics on the clone ratio [26]. Examining 136 versions of the kernel they found that the amount of clones was proportional to the system size, whereas most clones were caused by one particular subsystem.

Aversano and colleagues expanded the work of Kim and colleagues by investigating the same software systems, but adding further patterns regarding the evolution of clones [2]. They investigated how fragments of the same clone class were maintained by differentiating between inconsistently changed fragments that are continuously maintained independently and those that are made consistent in a later version (late propagation). It was found that most clone classes are maintained consistently, whereas the majority of inconsistent changes to clone classes were intended.

Göde presented a model for describing the evolution of cloned fragments and integrated it into the incremental clone detector iClones6 [10]. Empirical data have been obtained by analyzing the evolution of identical clones in nine open-source systems. It was found that the clone ratio decreased in the majority of the systems and that cloned fragments existed more than a year on average. Moreover, he found that either consistent or inconsistent changes to clone classes were more frequent depending on the system.

Saha and colleagues expanded the work of Kim and colleagues [21] by studying different aspects of clone genealogies at release level [31]. They analyzed 17 open source systems covering four different programming languages. Their results show that the majority of detected clones were either not changed or changed consistently and that many genealogies remain alive during the evolution.

In another study Saha and colleagues [32] evaluated clone genealogies of three open-source projects. They manually analyzed many of the detected genealogies considering predefined change patterns and conclude that their approach is scalable while maintaining high precision and recall.

We replicated and extended the study by Göde [10] by also considering near-miss clones [6]. By analyzing seven open-source systems we found that the clone detector reported remarkably more cloning related to near-miss clones compared to identical clones.

B. Changing Clones

Jarzabek and colleagues performed a case study on cloning in the Java Buffer library [18]. They propose a technique to unify clones in situations in which it is difficult to eliminate them with conventional program design techniques. The approach has been evaluated in qualitative and quantitative ways as well as in an controlled experiment. It was found that unifying clones reduced conceptual complexity and enhanced the changeability at rates proportional to code size reduction.

1A clone class contains all clone fragments that are sufficiently similar to each other.
Bakota and colleagues investigated change patterns in clone evolution using an AST based machine learning approach [3]. They used different similarity metrics to map individual cloned fragments of consecutive versions. Different patterns have been found that were related to bugs.

Krinke performed two separate studies to investigate the stability of exact clones [23], [24]. In the first study he analyzed five open-source systems looking at weekly snapshots over a 200-week period of time. He found that clones are changed inconsistently about half of the time, but late propagations are rare. In the second study he focused on whether cloned code is more stable than non-cloned code. Göde and Harder partially replicated and extended this study [12]. In general both found that cloned code is more stable than non-cloned code except for code deletions. In addition, Göde and Harder observed that varying parameters of the clone detector do influence the results, still conserving the relation.

Bettenburg and colleagues investigated changes to clones at release level focusing on inconsistencies [8]. In a case study on two open-source systems they observed that the risk of unintended inconsistencies is low and that only a very small number of inconsistent clones introduced defects.

Thummalapenta and colleagues conducted a study on four open-source systems to analyze specific patterns in the evolution of code clones and focused on examining characteristics of the later propagation pattern [33]. It was found that occurrences of the later propagation pattern were often related to defect-correcting changes. Barbour and colleagues expanded this study by using clone genealogies from two open-source systems to examine more characteristics of late propagations [5]. They found that the later propagation pattern indicates a risky cloning behavior regarding defects.

Another case study towards the correlation of late propagations and defects was conducted by Hui Mui [17]. Analyzing four Java systems based on log information provided by software repositories it was found that late propagations are not very common and about one quarter of the detected ones cause a bug—concluding that the overall impact is moderate.

Göde presented the first study of clone removals [11]. He investigated different aspects of deliberate clone removals to get a clue of the developers’ view on clones. Analyzing four open-source systems he found a number of intentional clone removals, but the gap to clones detected by a clone detector was remarkable though. Moreover, it was found that the scope of the refactorings hardly ever matched the scope of detected clones indicating that the programmers lacked awareness of the extent of clones in the projects or that clone tools are not accurate enough. Regarding metrics that might be useful to detect duplications that are good candidates for refactorings the findings did not provide clear results. Finally, Göde analyzed the committers of clone removals and found that the less people were involved in the projects, the more intentional removal of code clones took place.

Göde and colleagues also studied the frequency and risks of changes to clones in the history of three subject systems [14]. They found that 12.2 % of the clones were changed more than once and that nearly 14.8 % of all changes were accidentally inconsistent. Based on the results they conclude that the history of clones provides important insights to determine their relevance regarding maintenance tasks. Göde and Harder confirmed their findings in a follow up study [13].

Volanschi presented a refactoring technique for code clones [35]. The method is supposed to guarantee strong safety while leaving the spectrum of refactoring techniques open, for instance, to manual interventions. Volanschi evaluated the approach prototypically on a subset of a real-world legacy asset concluding that the results are promising.

III. ANALYZING CLONE REMOVALS

This section describes the semi-automated detection and the analysis of deliberate and accidental removal of code duplication in the evolution of a software system. The approach proposed in this paper can be separated into different steps that are repeated for each version of the subject systems. Each step will be described in the following. An overview of the framework is given in Figure 1.

A. Repository Mining

The Version Provider extracts necessary information from the particular software repository—currently Subversion and Git are supported. Only source files and corresponding log information are considered that match the programming language of the subject system under study (e.g., property and documentation files are ignored). Due to the incremental approach used in our clone detector, it is not necessary to analyze each version from scratch. Instead, source code changes of consecutive versions are determined and processed. After analyzing and extracting relevant files the Version Provider passes the fetched information to iClones.

B. Clone Detection

A clone detector is used to analyze all versions of the software system under study. The proposed framework in this paper uses an enhanced version of the clone detector used by Göde [11], namely iClones. iClones detects clones in two separate steps and, afterwards, maps clones between consecutive versions to build the evolution model for each

Fig. 1. iClones detects all clones in the evolution of a system’s source code provided by the Version Provider and writes the clone data in an RCF file. Afterwards, Cyclone reads the RCF to track and visualize the clone data based on an evolution graph.
system [6], [10]. First, all type-1 clones (code fragments without any differences) of the current version are detected by the **Clone Detector**. In the second step these clones are merged to larger near-miss\(^5\) clones by the **Clone Merger**, if possible. The algorithm is able to merge multiple adjacent code fragments to larger near-miss clones. To be merged, each code fragment has to be at least 10 tokens in size and the number of different tokens between two fragments needs to be smaller in size than the shorter of the two fragments. In addition, **iClones** has been configured to report only identical and near-miss clones with a minimum total length of 50 tokens. The values of 10 and 50 tokens for the minimum length thresholds is based on our experience from former studies using **iClones** [6], [10], [11], [15]. Figure 2 illustrates these bounds for identical and near-miss clones. Having detected all clone fragments of a version, every fragment is grouped with its clones into a clone class.

The detection process is repeated for each version of a program. During this process the **Clone Mapper** component uses a diff-based approach to map each fragment of version \(v_n\) to its ancestor fragment in version \(v_n+1\) by determining applied source code changes between the versions. By tracking each fragment throughout the history of the subject systems our clone evolution model is built, which enables analyzing clone fragments and their attributes independently over time. Due to its technical nature the mapping process is not described in detail in this work—we refer to the original publication for details on the mapping [6].

After analyzing every version of a system **iClones** writes the clone data into an **RCF** file that is used by our clone inspection tool **Cyclone**\(^6\) to detect clone removals in the next step.

### C. Clone Removal Identification

Detecting clone removals is quite difficult and time consuming as there is no method to do the task completely automated. However, G"ode [11] presented an approach that is intended to help detecting duplicated code that has been removed between two consecutive version \(v_n\) and \(v_n+1\) of a software system. We adapted this method to detect deliberate and accidental clone removals and integrated the approach into our clone inspection tool **Cyclone**. The identification of removals consists of two steps. The first is to filter all clones that cannot be part of any removal and the second is to manually decide which of the left-over clones are in fact affected by removals of duplication.

**Filtering:** The data generated by **iClones** contains all relevant data including the required information to build the evolution model. Therefore, **Cyclone** is solely used to inspect the huge amount of data computed by **iClones** and has no impact on the detection of cloned code, and hence, on precision and recall of the results. **Cyclone** is used to visualize the evolution model and helps the user to inspect different aspects of cloned code over time. To identify the removal of cloned code, the first objective is to determine the set of clone fragments that have been either modified or deleted. Cloned fragments that have not been changed can be ignored. In addition, fragments are filtered that have only been marginally changed between two versions of a system and, therefore, can most probably be neglected from further considerations. **Cyclone** offers a threshold that specifies the minimum bound of modified and deleted tokens in a cloned code fragment. In this study we used the same bound of 15 tokens as G"ode did. His study showed that this is an appropriate bound to filter many uninteresting changes with only a small risk to miss true removals [11]. Let \(F\) be the resulting set of clone fragments that have been changed sufficiently. It is notable that it is not sufficient to consider only clone fragments that have been completely removed from \(v_n\) to \(v_n+1\) as the scope of a refactoring hardly ever matches the scope of the detected clone fragments precisely.

The set \(F\) for each system is automatically detected by **Cyclone** based on source-code changes between consecutive versions determined using a standard diff algorithm\(^7\). Every clone fragment that is part of \(F\) is marked in the evolution model and the user is able to navigate through the set.

**Decision Process:** To decide from which fragments of the set \(F\) code duplications have been removed either deliberately or accidentally and whether possible refactorings have been missed by the developers must be done manually. For each fragment of \(F\) we checked the commit messages and reviewed the source code before and after the corresponding changes to judge whether code duplication has been removed or not. To assist the manual examination of commit messages **Cyclone** marks versions in the evolution graph whose commit messages include indicating keywords, e.g., removal, refactoring or duplication. Depending on the commit messages we sometimes had clear indications that code duplication was removed and where these changes took place exactly. However, from our experience commit messages are often imprecise or inaccurate so we used them only to get a first impression of what was done and not as decisive criterion. This means that we always analyzed the source code changes even if the commit message already gave a clear hint or gave no hint at all.

Afterwards, we analyzed the source code changes between two versions to get more reliable information. The code review was done on different levels. First, we used the integrated

---

\(^5\)We summarize type-2 (differences regarding identifiers and literals) and type-3 (differences apart from identifier and literals) clones as near-miss clones.

\(^6\)http://www.softwareclones.org

\(^7\)http://code.google.com/p/google-diff-match-patch/
source-code view of Cyclone, which focuses on the cloned fragments. It gives us the files, the exact positions of the cloned fragments and for near-miss clones the identical and non-identical parts at a glance. Especially, the non-identical parts of near-miss clones have to be kept in mind further on to separate changes to them from changes to the actual cloned parts. Due to the fact that our evolution model is based on clone classes, detected clone fragments might appear in more than one clone class of the same version, for instance, if fragments of different clone classes partially overlap the same code segments. Therefore, we used information about the fragments’ file paths and source locations for grouping overlapping fragments of different clone classes to save us from analyzing changes to the same code multiple times.

For a detailed code review we used the visual diff tool Meld\(^8\) and Eclipse\(^9\). Based on the information collected using Cyclone we used Meld to extract and review all modifications of files affected by changes that might have led to a removal of code duplication. In some cases we were not able to decide whether or not cloned code has been removed based on the change information provided by Meld. In such cases it was necessary to review more general project attributes in addition to the Meld diff, for instance, library and API updates. We used Eclipse to review the corresponding project attributes before and after the changes happened.

If we were able to detect the elimination of code clones by a refactoring, we exported the information of the corresponding clone fragments using Cyclone. The export is done automatically and collects different kinds of data. Among others we have exported basic information such as the version in which the refactoring took place, the file paths and source locations of the affected fragments and the date of change. In addition, we also exported different metrics, for instance, LOC, the number of consistent and inconsistent changes to the clone fragments before the refactoring and the Cyclomatic Complexity of the fragments. These metrics are used in our case study to investigate whether or not a ranking of clones can be based on these metrics. Such a ranking is needed to reduce the amount of clone data delivered to the users of clone management tools. If, for example, the number of changes to a certain clone fragment was high before it was removed, it might be an indication that the developers wanted to avoid the extra effort of continuously changing the cloned code and decided to refactor the corresponding source code.

**Categorization and grouping:** Based on this decision process we manually identified refactorings that removed code duplication and categorized the observed code removals. A schematic illustration of the resulting categories is shown in Figure 3. We started our categorization based on the initial set \(F\). In the first step we split \(F\) into the sets \(\text{Miss}\) and \(F'\). \(\text{Miss}\) includes all clone fragments related to refactorings that removed code duplication but missed some clones that could have been removed by the refactoring, too. Accordingly, the set \(F'\) includes clone fragments related to refactorings that removed all suitable clones. We further split the set \(F'\) into the two sub-sets \(\text{Del}\) and \(\text{Acc}\). \(\text{Del}\) includes clone fragments that were deliberately removed using appropriate refactorings. Figure 4 provides an example taken from Ant to illustrate our decisions on deliberate clone removals. Besides refactorings that were mainly performed to remove code clones, we also identified refactorings that removed code duplication accidentally as a side effect. This means that we found clear indications that the goal of the corresponding changes was actually not to remove clones. Clone fragments related to such refactorings are assigned to the set \(\text{Acc}\). We encountered different kinds of refactorings removing duplication as side effect, for instance, the removal of deprecated or dead code and updates of libraries and APIs. Note that clone fragments related to a refactoring that missed suitable clones are assigned to set \(\text{Miss}\) independent from the fact whether the refactoring was meant to remove duplication or not. The number of missed clones in this set is a helpful indicator when it comes to the question how much developers can benefit from using clone management tools that could pinpoint to similar code during code editing.

Finally, we grouped clone fragments within the sets \(\text{Del}\), \(\text{Acc}\) and \(\text{Miss}\) further. Clone fragments within the same set that are part of the same refactoring activity were counted as a ROD (Removal of Duplication) unit. The term ROD is adapted from Göde [11] and denotes a set of changes to remove duplication in which all clone fragments affected by one or more refactorings originated from a common intention, for instance, extracting equal functionality of a class into an method. RODs are manually identified and may contain fragments affected by refactorings that have been performed in different versions—corresponding to different commits—of a system as long as all of them have been carried out for the same reason. As an example, if three clone fragments

---

\(^8\)http://meldmerge.org/

\(^9\)http://www.eclipse.org/
were affected by the removal of code duplication during the introduction of a new utility class that includes a method providing the same functionality as the three deleted fragments and the fragments have been replaced by a method call to the new class, we count the removed fragments as one ROD. It does not matter whether all fragments have been replaced by the method call in the same or in different commits. Note that RODs can even have only one clone fragment assigned. For instance, if two statement sequences $A$ and $B$ are clones and $B$ is the body of a function $f$. If $A$ is replaced by a call to $f$, $A$ would be the sole fragment in the corresponding ROD.

IV. Case Study

In this section, we present the results of our evaluation. We analyzed eleven realistic open-source systems to gather empirical data that supplement previous findings and answer our research questions.

A. Study Setup

We selected the open-source systems based on the following criteria:

- The systems are open-source and maintained using a publicly available Subversion or Git repository.
- Each system has a reasonable size to provide a sufficient amount of data to obtain meaningful results, but is still manageable for manual inspection.
- We included systems that have already been analyzed by Göde [11] to allow for comparisons to his results.

The systems selected are Ant, FileZilla, FindBugs, FreeCol, Apache’s httpd, JabRef, Nautilus, Umbrello, ADempiere-Client, ArgoUML and TortoiseSVN. The last three were investigated by Göde [11], which we did not because KDE-Utils is a collection of different tools. Removing code duplication within one software system compared to the removal of clones between different systems is a different use case and brings in other aspects that need to be considered. The tools of KDE-Utils are related but though have their own source code and, therefore, present a combination of intra-system and inter-system clone detection use case. We investigated two years of each system’s history, starting from January 2009 to December 2010. We chose a time period that subsumes the time period investigated by Göde [11], which was from January 7 to October 29, 2009. We decided to increase the time period under study, because Göde reported that he had to dismiss different software systems from investigation as he was not able to find indications of clone removals. To lessen the risk of having the same problem and to check the impact the time period under study has on the results, we analyzed a longer period of the evolution of the subject systems. Snapshots have been analyzed on an interval of one day. Dates that did not contain any commits including changes to the source code were skipped. Details of the subject systems are given in Table I.

We have configured iClones to analyze the subject systems and report clone fragments with a overall minimum length of 50 tokens and a minimum length of 10 tokens for identical parts regarding near-miss clones as described in Section III-B. The reported output of each analysis was analyzed using Cyclone. In total we inspected 5152 clone fragments in 2967 clone classes that have been assigned to the set $F$ of cloned code fragments, using our approach and counting overlapping fragments that appeared in different clone classes of the same version just once. From these, we found 1293 fragments to be directly related to either deliberate, accidental, or missed removals of duplication. Unifying fragments that were related to the same refactoring task we identified 224 RODs.

To ensure the accuracy of the manual inspection and lessen the impact of subjectivity of the oracle, the two authors performed the manual inspection separately from each other. The first author did the manual analysis as described in Section III, before the second author checked 20% of the results for each subject system under study in a sampling process using the same approach as the first one did—not knowing the decision of the first author. The result of the sampled cross-check uncovered only few differences regarding the assessment of the refactorings performed, for instance, the scope of refactorings compared to the scope of detected clone fragments. The disagreements were all minor and could be resolved in short discussions among both authors. Further, we validated our results as far as possible using the study from Göde [11]. For those systems that have been covered and for those aspects that have been investigated by both studies we also found a general agreement regarding our results.

B. Removal of Duplication

Table II shows the results of our manual inspection and partially answers our research questions 1–3. The second and third columns show the number of clone classes and clone fragments detected by iClones that are related to RODs. The number of resulting RODs is given in column four. It can be seen that the numbers vary for each system with the number of detected RODs within the corresponding set of clone fragments. The overall numbers of RODs generally comply with the numbers of affected clone classes and fragments. Again, the most activity regarding the removal of code clones were found in TortoiseSVN. Except for TortoiseSVN, ArgoUML and FreeCol the number of removals in the other systems is rather

<table>
<thead>
<tr>
<th>System</th>
<th>Language</th>
<th>KSLOC</th>
<th>Clone Classes</th>
<th>Clone Fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADempiere-Client</td>
<td>Java</td>
<td>68 – 63</td>
<td>1,908,109</td>
<td>5,010,007</td>
</tr>
<tr>
<td>Ant</td>
<td>Java</td>
<td>175 – 160</td>
<td>696,446</td>
<td>1,813,720</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>C++</td>
<td>181 – 259</td>
<td>873,485</td>
<td>2,249,690</td>
</tr>
<tr>
<td>FileZilla</td>
<td>C++</td>
<td>103 – 113</td>
<td>513,912</td>
<td>1,150,498</td>
</tr>
<tr>
<td>FindBugs</td>
<td>Java</td>
<td>160 – 216</td>
<td>260,678</td>
<td>628,940</td>
</tr>
<tr>
<td>FreeCol</td>
<td>Java</td>
<td>94 – 108</td>
<td>614,193</td>
<td>1,605,787</td>
</tr>
<tr>
<td>httpd</td>
<td>C</td>
<td>187 – 200</td>
<td>747,684</td>
<td>1,734,741</td>
</tr>
<tr>
<td>JabRef</td>
<td>Java</td>
<td>80 – 103</td>
<td>347,460</td>
<td>888,642</td>
</tr>
<tr>
<td>Nautilus</td>
<td>C</td>
<td>136 – 155</td>
<td>330,957</td>
<td>807,891</td>
</tr>
<tr>
<td>TortoiseSVN</td>
<td>C++</td>
<td>150 – 260</td>
<td>1054,250</td>
<td>2,438,919</td>
</tr>
<tr>
<td>Umbrello</td>
<td>C++</td>
<td>200 – 202</td>
<td>734,862</td>
<td>1,774,858</td>
</tr>
</tbody>
</table>

IV. CASE STUDY

In this section, we present the results of our evaluation. We analyzed eleven realistic open-source systems to gather empirical data that supplement previous findings and answer our research questions.

A. Study Setup

We selected the open-source systems based on the following criteria:

- The systems are open-source and maintained using a publicly available Subversion or Git repository.
- Each system has a reasonable size to provide a sufficient amount of data to obtain meaningful results, but is still manageable for manual inspection.
- We included systems that have already been analyzed by Göde [11] to allow for comparisons to his results.

The systems selected are Ant, FileZilla, FindBugs, FreeCol, Apache’s httpd, JabRef, Nautilus, Umbrello, ADempiere-Client, ArgoUML and TortoiseSVN. The last three were investigated by Göde [11], which we did not because KDE-Utils is a collection of different tools. Removing code duplication within one software system compared to the removal of clones between different systems is a different use case and brings in other aspects that need to be considered. The tools of KDE-Utils are related but though have their own source code and, therefore, present a combination of intra-system and inter-system clone detection use case. We investigated two years of each system’s history, starting from January 2009 to December 2010. We chose a time period that subsumes the time period investigated by Göde [11], which was from January 7 to October 29, 2009. We decided to increase the time period under study, because Göde reported that he had to dismiss different software systems from investigation as he was not able to find indications of clone removals. To lessen the risk of having the same problem and to check the impact the time period under study has on the results, we analyzed a longer period of the evolution of the subject systems. Snapshots have been analyzed on an interval of one day. Dates that did not contain any commits including changes to the source code were skipped. Details of the subject systems are given in Table I.

We have configured iClones to analyze the subject systems and report clone fragments with a overall minimum length of 50 tokens and a minimum length of 10 tokens for identical parts regarding near-miss clones as described in Section III-B. The reported output of each analysis was analyzed using Cyclone. In total we inspected 5152 clone fragments in 2967 clone classes that have been assigned to the set $F$ of cloned code fragments, using our approach and counting overlapping fragments that appeared in different clone classes of the same version just once. From these, we found 1293 fragments to be directly related to either deliberate, accidental, or missed removals of duplication. Unifying fragments that were related to the same refactoring task we identified 224 RODs.

To ensure the accuracy of the manual inspection and lessen the impact of subjectivity of the oracle, the two authors performed the manual inspection separately from each other. The first author did the manual analysis as described in Section III, before the second author checked 20% of the results for each subject system under study in a sampling process using the same approach as the first one did—not knowing the decision of the first author. The result of the sampled cross-check uncovered only few differences regarding the assessment of the refactorings performed, for instance, the scope of refactorings compared to the scope of detected clone fragments. The disagreements were all minor and could be resolved in short discussions among both authors. Further, we validated our results as far as possible using the study from Göde [11]. For those systems that have been covered and for those aspects that have been investigated by both studies we also found a general agreement regarding our results.

B. Removal of Duplication

Table II shows the results of our manual inspection and partially answers our research questions 1–3. The second and third columns show the number of clone classes and clone fragments detected by iClones that are related to RODs. The number of resulting RODs is given in column four. It can be seen that the numbers vary for each system with the number of detected RODs within the corresponding set of clone fragments. The overall numbers of RODs generally comply with the numbers of affected clone classes and fragments. Again, the most activity regarding the removal of code clones were found in TortoiseSVN. Except for TortoiseSVN, ArgoUML and FreeCol the number of removals in the other systems is rather
small and for three systems we did not discover any removal in some change categories. Nonetheless, comparing deliberate and accidental removals, it can be observed that they are relatively balanced. The only striking gap is for Nautilus for which no deliberate, but 22 accidental RODs were found. This suggests that for the systems under study active clone management has been performed by the developers, but clones have also been managed unknowingly as a side effect of other refactoring activities. Regarding code clones that have been missed by suitable refactorings, we were able to detect only a few, except for TortoiseSVN. The results show that there may be an opportunity for clone detectors to reduce the likelihood of missed refactorings.

Overall, we can confirm the results of Göde’s study on deliberate clone removals for the software systems he studied [11], too, for the shorter time frame. Göde reported for ArgoUML one and for TortoiseSVN four more deliberate removals for the time period shared by both studies. The reason for the differences is that we classified these RODs as accidental removals. We assume that our more differentiated categorization compared to Göde’s contributed to this divergent judgment. Göde did not further classify RODs. Our differentiation may have led us to judge more strictly on deliberate removals.

C. Refactorings

To investigate how clones are removed by developers and complete the answers to our research questions 1–3, we have investigated the refactorings related to the detected RODs presented in Section IV-B. A better understanding of which clones attract the attention of developers might help to improve existing detection and management tools to produce more useful results by ranking the results. Our case study setup suits the use case of removing code duplication and therefore we inspect what kind of refactorings developers used to remove clones and how well the scope of the refactorings fits the scope of the clone fragments reported by our clone detector. Göde inspected refactorings and their scope in his previous study [11]. He stated that a good and comprehensive matching is the basis for the use of clone management tools. On the contrary, a bad matching indicates either automated detection tool and duplications removed by developers—the scope of the refactorings hardly matched the scope of the detected clone fragments.

We adapted the approach to categorize RODs from Göde [11] to our results. The taxonomy we used to classify refactorings is an extended version of his taxonomy and emerged as a result of our manual inspection process to detect refactorings that removed code clones. Our classification is based on refactorings commonly quoted as suitable to remove duplications, for instance, Extract Method and Pull Up Method. Figure 5 shows the resulting classification. The bold rectangles depict which parts have been added to the classification compared to the one presented by Göde.

We detected three categories of refactorings that were applied to remove clones. In contrast to Göde we do not separate unifications into a category on its own, because gathering code that provides equal functionality in a single place is based on the movement of the corresponding code and, therefore, we count unifications in our Movement category.

**Replacement** We distinguish two cases in this category. The first one is that a sequence of statements has been replaced by a single method call either to a newly introduced method (New) or to a method that already existed before the refactoring took place (Exist). The second one includes the replacement of a sequence of statements within methods by another sequence of statements (Modify). In most cases we detected that the nesting within methods has been reduced due to high complexity—which also led to the removal of clones.

**Movement** This category includes refactorings—such as pull-up field or method—that gather two or more field declarations or methods providing equal functionality in a single class. We differentiate between three types of movements. First, the code is moved to a new class that did not exist before (New). Second, the code of the affected classes is moved to a superclass (Super) of them. Last, the code is moved to an existing class that provides functionality used by a range of classes, e.g., a utility class, but that class is not a superclass of the classes from which the code is moved (Other).

**Deletion** Field declarations or methods are removed. Mainly this category includes deletions of dead or deprecated code. We categorized deletions of deprecated code as deliberate clone removals if it was still in use before the corresponding refactoring deleted it and the refactoring’s
TABLE III

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>RODs</th>
<th>Match</th>
<th>Containing</th>
<th>Contained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacement</td>
<td>39</td>
<td>10</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>New</td>
<td>31</td>
<td>9</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>Modify</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Exist</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Movement</td>
<td>45</td>
<td>5</td>
<td>9</td>
<td>30</td>
</tr>
<tr>
<td>New</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Super</td>
<td>33</td>
<td>5</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>Other</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Deletion</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>86</td>
<td>17</td>
<td>26</td>
<td>43</td>
</tr>
</tbody>
</table>

In addition to the taxonomy, we adapted the approach of comparing the scope of refactorings to the scope of clone fragments reported by our clone detector from Göde [11]. For each ROD the scope of its clone fragments is manually compared to the related refactoring performed. An advantage of the manual comparison is that we can tolerate various artifacts of token-based clone detection at the beginning and end of fragments, for instance brackets. The comparison results in each fragment being classified to either:

- **Match**: if all statements of the clone fragment are part of the refactoring and vice versa
- **Containing**: if the changes refer only to statements within the bounds of a clone fragment
- **Contained**: if the scope of the refactoring (the statements changed by the refactoring) subsumes the clones either completely or partially, that is, just overlaps with the clone fragment

We found that for all clone fragments in the same ROD the comparison of scopes led to the same classification because of which we report the resulting numbers based on RODs rather than on clone fragments. Table III presents the results of deliberate RODs classified as either **Match**, **Containing** and **Contained**. Although, we have analyzed accidental and missed RODs and their corresponding scope as well, we leave out the exact numbers, because these RODs were not meant to remove code duplication in the first place. It is arguable whether the scope of such refactorings needs to match the scope of the clone fragments at all. However, analyzing the results for accidental and missed RODs, we generally found the same trend as presented for deliberate RODs. Note that the two categories **Modify Existing Method** and **Move to Existing Class** of our classification were added because of refactorings related to accidental or missed removals of code clones. There were no instances of deliberate removals in these categories.

Table III shows that our results confirm the large discrepancy between the scope of detected refactorings and the scope of detected clone fragments stated by Göde [11]. Overall only about 20% of all refactorings have been categorized to match the scope of the detected clone fragments. Göde found 16% of his analyzed refactorings to match the scope of clone fragments. Focusing on refactorings that did not match the scope of detected fragments we found that for the category **Movement** clearly more refactorings go beyond the scope of clone fragments. In contrast to that, there is no trend for refactorings that replaced source code regarding how well their scope fits the scope of detected clones. Looking at refactorings that replace or move code, respectively, their occurrence is almost balanced. In comparison, deliberate clone removals by refactorings that fall into the category **Deletion** are quite rare.

Finally, we analyzed how many of the clone classes that included clone fragments affected by deliberate or accidental removals of duplication disappeared completely in the version after the corresponding refactoring was applied. We found that about 90% of the clone classes affected by deliberate clone removals and about 80% of the clone classes affected by accidental removals disappeared in the following version. This could be assumed to prove a high success rate in reducing clones regarding the refactorings chosen by developers. However, it should be kept in mind that the amount of clone classes and fragments related to the reduction of duplications is quite small compared to the overall numbers of clone classes and fragments detected by our clone detector.

D. Ranking Clones

To investigate whether and what measurable characteristics may help in ranking clone candidates for removal and answer our research question 4, we collected different clone metrics based on our retrospective analysis. Göde investigated different aspects of clone removals to contribute to the ranking of clones regarding the use case under study [11]. He limited his study to the following attributes: length, similarity, distance in the source tree, and number of source code files that contain the fragments of a ROD. Overall his results did not yield clear results, only for the distance attribute there was a trend that developers mostly removed duplicated code located in the same source code file. Göde assumed that other attributes may help in ranking clones for removal. To contribute and extend Göde’s preliminary analysis we collected and evaluated different clone characteristics based on the clone classes and clone fragments related to our detected RODs.

The first aspect we investigated is whether developers tend to perform more refactorings on code clones that are identical or just similar. We analyzed how often RODs were related to identical and how often to near-miss clones based on detected clone classes rather than on clone fragments. This is sufficient as all clone fragments of the same clone class have the same clone type, because clone fragments are grouped in clone classes based on their level of similarity. Table IV shows the result considering type-1, type-2, and type-3 clones. The first column depicts the three sets of refactorings we have investigated, the second column the number of clone classes affected by removals of duplication and the last three columns depict how many of these clone classes have been assigned to either type-1, type-2, or type-3 clones.

Göde used the term Superior instead of Containing and Inferior instead of Contained—we considered the terms Superior and Inferior mistakable.
Cyclomatic Complexity column gives the average related to RODs before they have been refactored. The last (FCC) and inconsistent (FIC) changes to clone fragments next two columns depict the maximum numbers of consistent of tokens. By how many tokens the clone fragments have clone removal has been performed. The second column gives column specifies whether a deliberate, accidental, or missed fragments that are good candidates for refactorings, too.

It was found that there is no real tendency towards one of the analyzed clone types. Each type roughly constitutes one third of all detected RODs independent from the classification of the refactorings. However, the numbers indicate that type-1 clones are more often deliberately removed and that type-2 and type-3 clones are slightly more often affected by accidental removals as a side effect. The results are a bit surprising as we expected a more distinct trend towards deliberate removal of identical clones, because a semantic preserving refactoring of near-miss clones naturally requires more effort and often there is no automated tool support that can be used to perform the refactoring. On the other hand, we assume differences of type-2 clones to be rather moderate what makes them close to identical and, hence, easier to refactor.

Apart from the clone type, we collected data regarding the length of fragments and how many fragments affected by RODs decreased in size. Regarding changes to clone fragments we have analyzed how often detected clone fragments have been changed consistently or inconsistently, respectively, over time until they have been refactored by developers. The change frequency of clone fragments may indicate which clones might be good candidates for a removal. Assuming that a high change frequency of cloned code causes additional effort to keep the clones synchronized, a high change frequency is costly for developers and a refactoring probably pays off in the long term as changes have to be done at only one single place afterwards. The last characteristic of clone fragments we looked into is the Cyclomatic Complexity. A high Cyclomatic complexity is an indicator for source code that has a more involved control flow. As a consequence changes need more effort to be performed and tested. Based on this assumption the Cyclomatic Complexity may be useful to identify clone fragments that are good candidates for refactorings, too.

Table V shows the results for the collected metrics. The first column specifies whether a deliberate, accidental, or missed clone removal has been performed. The second column gives the average size of the affected clone fragments in number of tokens. By how many tokens the clone fragments have been decreased on average is given in the third column. The next two columns depict the maximum numbers of consistent (FCC) and inconsistent (FIC) changes to clone fragments related to RODs before they have been refactored. The last column gives the average Cyclomatic Complexity (CC) of clone fragments related to RODs.

The results show that the clone fragments related to RODs are clearly reduced in size by the performed refactoring. For deliberate and accidental removal of duplication we have measured an average decrease around 70% of the fragment’s size. Regarding partially missed fragments we have an average of nearly 50% decrease. Analyzing how frequent the affected clone fragments have been changed before the removals took place, we detected very few changes at all. Looking at the maximum number of consistent and inconsistent changes to clone fragments, we see that they range from 0 to 3. On average less than 1% of clone fragments related to RODs changed before the detected refactorings—no matter whether consistent or inconsistent. These numbers are rather small and, therefore, we assume that the change frequency of clone fragments cannot be used for an automated ranking of clones. Analyzing the Cyclomatic Complexity we detected a diverse distribution between fragments. The absolute numbers range from 1 to 35 without any clear tendency. On average the Cyclomatic Complexity ranges from 2.7 up to 5.2 per fragment as shown in Table V. Because of the large diversity among clone fragments, we could not identify any pattern to rank clones using the Cyclomatic Complexity.

E. Threats to Validity

Tools The accuracy of our results depends on recall and precision of iClones. iClones uses a token-based detection technique and token-based techniques are considered to provide high recall and reasonable precision [7] and that the manual inspection of the reported data reduced the threat of low precision heavily.

Subjectivity Our results are partially exposed to our subjective assessment of clones and refactorings. To mitigate the threat, the manual analysis was first performed by one of the authors and, afterwards, the second author independently checked a sample of 20% of the findings to verify the results.

Subject Systems We based our study on eleven open-source systems from different domains. Yet, this sample may still not be representative. Only a part of their history has been analyzed. Using a different period of time or analyzing snapshots at a different interval might affect the findings.

V. Conclusion

We adapted the semi-automated approach from Göde [11] to detect removals of duplication in the history of a software system and integrated it into the clone inspection tool Cyclone. Based on this approach we tracked individual clones of eleven open-source systems over a period of two years investigating cloned code to answer our four research questions.

To answer our first research question, we analyzed deliberate clone removals and were able to find occurrences of removals in all systems except for Nautilus and Umbrello. The applied refactorings mainly replaced existing code by calls

---

**TABLE IV**

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Classes</th>
<th>Type-1 [%]</th>
<th>Type-2 [%]</th>
<th>Type-3 [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Del</td>
<td>331</td>
<td>37.0</td>
<td>28.0</td>
<td>35.0</td>
</tr>
<tr>
<td>Acc</td>
<td>247</td>
<td>32.3</td>
<td>29.7</td>
<td>38.0</td>
</tr>
<tr>
<td>Miss</td>
<td>87</td>
<td>29.0</td>
<td>32.3</td>
<td>38.7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>665</td>
<td>32.8</td>
<td>30.0</td>
<td>37.2</td>
</tr>
</tbody>
</table>

**TABLE V**

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Token [∅]</th>
<th>FCC</th>
<th>FIC</th>
<th>CC [∅]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Del</td>
<td>112.1</td>
<td>65.2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Acc</td>
<td>114.2</td>
<td>72.1</td>
<td>2</td>
<td>4.7</td>
</tr>
<tr>
<td>Miss</td>
<td>74.4</td>
<td>46.0</td>
<td>1</td>
<td>2.7</td>
</tr>
</tbody>
</table>
to newly introduced methods and gathered common code of specialized classes in their superclass. Investigating accidental clone removals and answering our second research question, we found that accidental removals of duplication occur slightly more often than deliberate ones. Moreover, the refactorings used are basically the same as used for deliberate removals. That is, there is a high chance that existing refactoring support can be used for clone management, too. Based on the overall similarity in the frequency and type of the applied refactorings that removed clones deliberately as well as accidentally, we conjecture that an integration contributes to the acceptance of clone management tools as integral part of the development. The answer to our third research question whether refactorings removing clones may be incomplete further supports this point. We detected situations in which refactorings missed some clone fragments that could have been removed, too. This observation suggests that further research should investigate whether developers aided by automated clone detection during refactoring remove clones more completely.

To answer the last research question we measured different characteristics of cloned code that have been removed to observe whether they help in ranking clone candidates for removal. Interestingly, more near-miss clones were removed in the systems than identical clones. At least, for identical clones we expected to detect more deliberate removals, because removing near-miss clones normally needs more sophisticated refactorings. Analyzing the change frequency in the evolution of clone fragments revealed that the removed clone fragments rarely changed before, contrary to our expectations. Finally, we did not find any relation between control flow complexity measured as Cyclomatic Complexity and clone removal.

In future work we plan to conduct a survey based on a questionnaire and interviews of the developers who performed the refactorings related to the RODs of our analysis to gain further insights into a developer’s view and how that knowledge can be used to rank code clones and improve existing tools.

ACKNOWLEDGMENTS

We thank Emil Baybulatov for his support on implementing technical features. This work was supported by a research grant of the Deutsche Forschungsgemeinschaft (DFG).

REFERENCES