# A Replication and Reproduction of Code Clone Detection Studies

**Xiliang Chen**[1] **Alice Yuchen Wang**[1] **Ewan Tempero**[2]

[1] Electrical and Computer Engineering
The University of Auckland
Auckland, New Zealand
xche185@aucklanduni.ac.nz,ywan478@aucklanduni.ac.nz

[2]Computer Science
The University of Auckland
Auckland, New Zealand
e.tempero@auckland.ac.nz

**Abstract**

Code clones, fragments of code that are similar in some way, are regarded as costly. In order to understand the level of threat and opportunity of clones, we need to be able to efficiently detect clones in existing code. Recently, a new clone detection technique, CMCD, has been proposed. Our goal is to evaluate it and, if possible, improve on the original. We replicated the original study to evaluate the effectiveness of basic CMCD technique, improved it based on our experience with the replication, and applied it to a 43 open-source Java code from the Qualitas Corpus. We confirmed the effectiveness of the original technique but found some weaknesses. We improved the technique, and applied our improved technique. We found that that 1 in 2 systems had at least 10% cloned code, not counting the original, indicating that cloned code is quite common.

## 1 Introduction

Code clones, fragments of code that are similar in some way, are regarded as costly (Juergens et al. 2009, Li & Ernst 2012). There is some evidence that the number of clones in any given system can be non-trivial (Baker 1995, Baxter et al. 1998, Falke et al. 2008, Juergens et al. 2009, Schwarz et al. 2012). This means, that if code clones do represent an unnecessary cost, then their existence is both a threat to the software quality and an opportunity for improvement. In order to understand the level of threat and opportunity, we need to be able to detect clones in existing code. If there are few code clones, then their cost is not so important. In order to understand the cost associated with clones, we need to be able to identify clones and determine the cost due to their existence.

To detect clones, we need a means that is both efficient and effective. If the techniques used to detect clones have poor accuracy, then effort will be wasted identifying false positives, and the results will be uncertain due to the unknown false negatives. If the clone detection techniques are inefficient, it will be difficult to gather enough data to be useful. Many existing clone detection techniques are quite expensive, particularly in time, or trade off time for accuracy.



Recently, Yuan & Guo (2011) demonstrated a new technique, CMCD (for Count Matrix Clone Detection), for detecting clones that is very efficient and appears quite effective. The basic idea is language-independent and relatively straight-forward to implement, so that it may be possible to produce good clone detectors for many languages quite quickly. In fact it is very simple, especially compared to other clone detection systems, raising questions as to whether it can be generally effective. If it is as good as it appears, CMCD has very good potential for significantly increasing the number and size of empirical studies, thus improving our understanding of the impact of code clones. While the paper explained CMCD well, the evidence it provided for how good CMCD is was quite weak, so there is still a question as to its effectiveness. In this paper, we answer these questions. Specifically, we confirm the original claims, present an improved version of the technique, demonstrate that it is effective, and present the results from using the new technique on a large corpus of Java code. These results indicate that about half the systems we studied had more than 10% cloned code.

In scientific study, it is not enough to observe something once to be convinced of the validity of some theory, the observations must be *repeated*. As Popper said, "We do not take even our own observations quite seriously, or accept them as scientific observations, until we have repeated and tested them." (Popper 1968) While there have been a number of empirical studies reporting the degree to which code clones exist, there are various issues that exist with those studies. Their goal is usually not to determine to what degree clones exist, but to demonstrate how effective a given clone detector is. They rarely examine the same systems as each other, so it is not obvious how to compare the results with each other. They also are generally quite small. We are aware of only 2 large studies, and they are both of systems written in C (Uchida et al. 2005, Kamiya et al. 2002). Many more studies are needed, and there is a need to perform large studies of other languages. Our work is another step in this process. In this paper we present one of the largest code clone detection studies to be undertaken in Java.

What constitutes a useful repetition is a matter of some debate (Drummond 2009), however in this paper we consider what Cartwright refers to as *replicability* — doing the same experiment again — and *reproducibility* — doing a new experiment (Cartwright 1991). In this paper we replicate (as much as possible) the study by Yuan and Guo to demonstrate the validity of CMCD and we attempt to reproduce the results of various empirical studies undertaken to de-

termine the degree to which code clones exist in Java code.

The rest of the paper is organised as follows. In the next section we present some of the literature on empirical studies of code clones to determine what has been established so far. In section 3, we summarise the original CMCD technique. In section 4, we describe the modifications we have made to CMCD, and how we carried out the replication and reproduction studies. We present our results in section 5 and discuss them in section 6. Finally we present our conclusions and discussion future work.

## 2 Background and Related Work

In this section we introduce the concepts generally associated with clone detection and give an overview of clone detection related research. The literature in clone detection research is quite extensive, and so we necessarily can only give a sample here. Roy et al. (2009) provide a good survey and we use their terminology below. We detail only the work that is directly relevant to ours.

A *code fragment* is any sequence of code lines that can be any granularity, such as a complete method definition or sequence of statements inside an if-statement (Bellon et al. 2007). A *clone pair* is defined by having two code fragments that are similar by some given definition of similarity. When more than two fragments are similar, they form a *clone cluster* or clone group. There are two main types of similarity between code fragments: textual similarity and semantic similarity. Two fragments have textual similarity when the text they contain matches to a large degree. This might be the consequence of copying and pasting one fragment to the other, perhaps with minor modifications. Two fragments are semantically similar if they have similar functionality, but may have completely different text. Clone types can be categorised into four types based on both textual and semantic similarities (Roy et al. 2009):

**Type-1:** Identical code fragments except for variations in whitespace, layout and comments.

**Type-2:** Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

**Type-3:** Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

**Type-4:** Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Most of the clone detection techniques can be summarised into four main categories: textual, lexical, syntactic and semantic (Roy et al. 2009).

Textual approaches compare the source code with little or no transformation. In most cases raw source code is used directly in the clone detection process. Such approaches must cope with variation of all kinds, including in whitespace. An early such approach was described by Johnson (1994).

Lexical approaches transform the source code into a sequence of lexical tokens using compiler-style lexical analysis. Comparison is then done between sequences of tokens to identify common subsequences. These approaches easily deal with variation in whitespace, layout, and comments. Also, variations in identifiers or literals can also be dealt with. An example of this approach was described by Baker (2007).

Syntactic approaches use a parser to convert the source code into parse trees or abstract syntax trees (ASTs). Comparison is then done between the trees for commonality, possibly using structural metrics. The work by Baxter et al. (1998) is perhaps the best known of these approaches.

Semantic approaches use static program analysis, which does a more sophisticated comparison than just at the syntax level. One technique that is used is to represent the code as a program dependency graph (PDG) and then analyse that. An example of this approach uses backward slicing (Komondoor & Horwitz 2001).

Roy et al. (2009) describe 4 *scenarios* giving examples of each of the categories described above, with further sub-scenarios for each category, a total of 16 examples. They then evaluated more than 40 techniques with respect to these 16 examples, providing both clear criteria (at least as a starting point) for what might constitute a clone, and a comprehensive summary of that the state-of-the-art at that time could handle.

As mentioned in the introduction, our work is based on the CMCD technique developed by Yuan & Guo (2011), which we will detail in the next section. This technique is a syntactic approach, specifically it falls in to the category Roy et al. call metrics-based approaches. Generally, these approaches make measurements of the code fragments using one or more metrics. The value of the measurements indicate how similar the code fragments are. One example of this is by Mayrand et al. (1996) who gather measurements from a number of metrics, such as different characteristics of the layout of the code, number and type of statement, and characteristics of control flow such as number of decisions, average nesting level, and number of exits. Another example is by Kontogiannis (1997), who uses more common metrics such as Fan-in, Fan-out, and Cyclomatic Complexity Number.

Once measurements are produced, the existence of clones is determined by the similarity of the measurements. The comparison of measurements may be done in different ways, depending on who the measurements look like. For example, Kontogiannis groups the measurements into 4 categories, and then compares each category separately. The results of the 4 comparisons are then used to produce an ordinal scale measurement capturing how different (or similar) two code fragments are. Another example is by Davey et al. (1995), who use a self organising neural net to do the comparisons.

With all the variations in techniques, the question that naturally arises is which is the best? Unfortunately there is no clear answer, not only because the answer depends on the reason for detecting clones, but also because there has been insufficient comparison between the techniques.

For an example on how context might affect which quality attributes of the clone detection technique we desire, consider plagiarism detection in student assignments. In this context, we would probably do this off-line (as a batch process), we might expect the size of the inputs to be relatively small (hundreds, or perhaps a few thousand, lines of code), and we would likely be willing to accept missing some cases (false negatives) in order to ensure not making any false accusations (false positives). On the other hand, to support a code review function in an IDE, we would want real-time information of possibly a large code base,

but would accept a reasonable degree of false positives and false negatives. These two examples represent a trade-off of preferences in performance versus accuracy. Other trade-offs include what constitutes a clone, for example only requiring detection of Type 1 and Type 2 clones, and what granularity of code fragments are to be considered, such as comparing only subroutines or comparing blocks.

As an example of other uses of clone detectors, Li and Ernst examined the degree to which code clones also contained duplicated buggy code (Li & Ernst 2012). Their detector used a semantic approach based on PDGs. They examined 3 systems (Git, Linux kernel, and PostgreSQL). Using the systems' bug reports, they identified the code fragments where the bugs occurred, and then tried to find clones of those fragments. They compared their system against 4 other clone detectors. Their system performed as well or better than the others in both accuracy and performance.

There have been some comparisons of different techniques. Bellon et al. (2007) compared 6 clone detectors that used different approaches over 4 *C* and 4 Java systems with respect to their accuracy and performance. While there was no clear winner, they noted that the AST-based approaches tended to have higher precision (fewer false positives) but were also longer execution time, whereas token-based approaches had higher recall (fewer false negatives) but were faster. They commented that "if idea from the token-based techniques could be made to work on ASTs, we would be able to find syntactic clones with less effort."

Falke et al. (2008) did a follow up study using the same infrastructure as by Bellon et al. to examine the quality of clone detectors based on suffix trees. They found that using suffix trees was faster than the standard AST matching, but with varying recall and precision.

Two important questions relating to clone detection research are: Is the belief that clones are a problem correct, and; Are there enough clones in real code to matter? Juergens et al. (2009) addressed the first question by developing a new clone detector and applying it to 5 projects, 4 from 2 companies (3 in C#, 1 in Cobol), and one open source (Java). They were particularly interested in what they called *inconsistent* clones, code fragments that differ by more than just simple changes that apply to the whole fragment, such as renaming variables (effectively Type-2 clones). They presented identified inconsistent clones to the developers of the systems, and from the developers determined whether the inconsistency was intentional or not, and whether the clones were faulty or not. From this they concluded that inconsistent clones are a major source of faults. The results by Li & Ernst (2012) also suggest that clones are a problem, by finding clones that contain the same bugs.

There does not appear to have been any systematic large-scale studies to determine the degree to which code clones exist. However, most presentations of new clone detectors provide data from their application that indicates that clones are relatively common.

For example, in an early study, Baker found on the order of 19% of the X Window System (Baker 1995) are either Type-1 or Type-2 clones. Baxter et al. looked at 19 subsystems of a process-control system with 400 KSLOC of C code (Baxter et al. 1998). Their results indicated that on average 12.7% of code was cloned, with at least two subsystems having over 28%.

Juergens et al. found 300 or more clone groups (2 or more code fragments that are clones of each other) in 4 of the 5 systems. They do not indicate what proportion of the overall code base these groups represent, nevertheless it must be non-trivial.

While Bellon et al.'s goal was to compare detectors, they include information about candidates reported by the tools, the size of code fragments identified as candidates, and information on accuracy in their results. While it is difficult to determine the proportion of cloned code, again it is clear that it must be non-trivial.

Schwarz et al. (2012) examine the repositories of Squeaksource, a repository in the Smalltalk ecosystem. They found 14.5% of all methods strings (560K different methods in 74K classes) were present in at least two distinct repositories.

We are aware of two large empirical studies, both by the same research group. The earlier one studies 125 packages of open source software written in C (Uchida et al. 2005). The size of the systems ranged between 478 and 2,678,939 "LOC". They do not report how they measured LOC. They found there is much variation in how much cloned code there is, but on average they found 11.3%. This group did a later study of the "Packages and Ports Collection" of FreeBSD. The primary goal of this study was to demonstrate their distributed version of CCFinder (Kamiya et al. 2002). They analysed more than 400 million LOC over nearly 6700 projects, which appeared to include many of the systems in their previous study. They found on average 4% of code clones, but there were several cases where the degree of code clones was much higher. However they did not report results for individual projects.

In summary, various studies consistently report that code clones exist to a non-trivial degree, with many measurements of more than 10% being reported. However, most studies are only of a small number of systems, and many of those systems are quite small. What large studies there are examine only systems written in C. Our interest is whether we would see different results in a different language (Java in our case).

## 3 Original CMCD Technique

In order to make our contribution clear a good understanding of the original CMCD technique is needed, which we provide below. More details are available in the original publication (Yuan & Guo 2011). The modifications we made are described in the next section.

The CMCD technique determines the similarity between two code fragments by modelling each with a *count matrix* and comparing the count matrices. A count matrix is made up of a set of *count vectors*. In the original CMCD, there is one count vector for each variable that appears in the code fragment. The values in a count vector come from a set of *counting conditions* that are applied to the variable that vector represents. The counting conditions represent how a variable is "used". The intuition is, if two code fragments are indeed clones, then a variable in one fragment will have a counterpart in the other fragment that is used in very similar ways, so the count vectors will be very similar. Also, most variables in one fragment will have counterparts in the other fragment, so the count matrices will be very similar. If, on the other hand, the fragments are very different, then there is a high probability that many variables in one fragment will have no obvious counterpart in the other, so the count matrices will look different.

Table 1: The original Counting Conditions (Yuan & Guo 2011)

| | |
|---|---|
| 1 | Used |
| 2 | Added or subtracted |
| 3 | Multiplied of divided |
| 4 | Invoked as parameter |
| 5 | In an if-statement |
| 6 | As an array subscript |
| 7 | Defined |
| 8 | Defined by add or subtract operation |
| 9 | Defined by multiply or divide operation |
| 10 | Defined by an expression which has constants in it |
| 11 | In a third-level loop (or deeper) |
| 12 | In a second-level loop |
| 13 | In a first-level loop |

As the Yuan and Guo noted in the original publication, exactly what constitutes a "use" is maybe not as important as applying the counting conditions consistently. Nevertheless the counting conditions do need to indicate some reasonable notion of "use". The original counting conditions are shown in Table 1. These counting conditions all are uses of variables that are familiar to any programmer. Clearly other counting conditions are possible as the authors acknowledge, but it is not obvious whether the extra cost of adding more will significantly change the outcome. We return to this point in the next section.

Two count vectors are compared by computing the normalised distance between them. The original technique uses euclidean distance and normalises (roughly) by dividing by the vector lengths (see paper for full details). The resulting distance is in the range [0..1], where 1 means identical.

After computing the count vectors for each variable for each code fragment, the resulting count matrices need to be compared to determine similarity. An issue arises in that, while each variable in one fragment may have a very similar counterpart in the other fragment, this may not be obvious if the order of the count vectors is different in the count matrices, that is, it is not enough to just compare the first row of one matrix with the first row of the other, and so on. CMCD resolves this issue using maximum weighted bipartite matching as follows.

Each row in the two matrices being compared is treated as a vertex in a graph, and each vertex from one matrix has an edge to every vertex in the other matrix. Each edge is weighted by the distance between the two respective count vectors. This results in a weighted bipartite graph. The maximum weighted bipartite matching of this graph is then a pairing of count vector from one matrix with a count vector in the other matrix that maximises the sum of the count vector distances. This sum is then the measure of similarity between the code fragments.

The similarity value may also be normalised, to account for comparing code fragments of different sizes, or have a different number of variables. Also, in case it is possible for two quite different fragments to get a high similarity measurement, a false positive elimination step is applied using heuristics. The authors do not give any details as to what heuristics they use.

The same idea can be used to compare two sets of code fragments — a weighted bipartite graph can be constructed where a vertex is the count matrix for a code fragment, and edges are between vertices from one set to the other weighted by the similarity score between the corresponding code fragments. Again, maximum weighted bipartite matching can be used to determine how similar the two sets are. In this way two classes can be compared for similarity by treating each method as a code fragment and applying the technique as described above.

Yuan and Guo evaluated their CMCD technique by using it in three different ways. First, they applied it to the 16 scenarios described by Roy et al., demonstrating that it detected all 16 cases. They then applied it to 29 student medium-sized project submissions (7 KLOC – 38 KLOC, 585 KLOC in total). The processing took 123 minutes on relatively standard hardware and they found 2 clone clusters. Manual examination concluded that would have been difficult to identify the clusters through manual inspection. Despite the fact that all projects implemented the same functionality, they did not find any false positives.

The third evaluation method was to analyse JDK 1.6.0_18 (about 2 MLOC). They compared every pair of methods in this code base, ignoring very small methods such as getters and setters. The processing took 163 minutes and found 786 similar methods over 174 clusters. One of the clusters included at least one instance that appeared to contain a fault. They provide no information of how the quality of these results was determined.

The evaluation provided by Yuan and Guo indicates that CMCD has some value, but only one large system was analysed, and it is difficult to just the quality of its results.

## 4 methodology

The research questions we would like to answer are:

**RQ1** Is the CMCD technique as effective as its authors claim and can it be improved?

**RQ2** How much code is created through cloning?

The basic steps we follow are:

1. Implement the CMCD technique as close as practical to the original.
2. Perform two of the three evaluations described in the original paper (see section 2).
3. Based on the results of, and experience gained by, performing the previous step, refine our implementation.
4. Evaluate the effectiveness of the refinement, returning to step 3 if the results indicate the need for, or possibility of, improvement.
5. Apply the refined implementation to a large body of code, returning to step 3 if the results indicate the need for, or possibility of, improvement.

Some of these steps are elaborate further below.

There are two details we need to clarify: what definition of clone we are using and what level of granularity of clone we will detect.

As others have noted, in particular Roy et al. (2009), there is no agreed upon evaluation criteria as to when two code fragments are clones. We use the same intuition as others, namely that two code fragments are clones if one could "reasonably" have resulted by copying and pasting the other and making "minor" changes. While this introduces a degree of subjectivity, we follow Yuan and Guo and use the scenarios proposed by Roy et al., which provides some means of comparison with other work. We discuss this further in Section 6.3.

We also follow the original CMCD technique, which compares code fragments that are methods,

that is, it does not detect clones that are smaller than methods. We choose to do so as one of our goals is to replicate Yuan and Guo's study. How this technique might be applied to sub-method clones is a topic for future work.

### 4.1 CMCD implementation

As Yuan and Guo used Java as their target language, we choose to do the same. Their implementation determined the count matrices based on the Jimple representation of the Java source, which is a 3-address code representation produced using the SOOT framework (Vallée-Rai et al. 1999). We had a concern about this decision.

The Jimple representation is necessarily different from the original source code, and furthermore is a transformation of the compiled code (bytecode) rather than the original source code. Yuan and Guo argue that these transformations have little effect on the results. Our concern is that the two transformations may mean that slight differences in the source code may result in more significant differences in the Jimple representation. For example, information could be lost during compilation which may affect the level of accuracy, especially if optimisation techniques are used. Also, the transformation to Jimple involves introduction of temporary variables, and slight differences in the source code may result in different temporaries, potentially resulting in a more significant change at the Jimple representation than exists in the original source.

If we are right, then we would get better results dealing with the source code directly. Furthermore, if Yuan and Guo are right, it should not matter if our implementation uses a different technique to determine the count matrices.

Consequently we decided to base our implementation on a parser for Java source code. We used ANTLR (`antlr.org`) to create the parser. This produces an Abstract Syntax Tree (AST), which is then traversed, applying the counting conditions as appropriate to each vertex. The count matrices are created and compared as in the original.

Unlike the original technique, rather than measure similarity between methods (smaller values means less similar), we measured *differences* (smaller values means more similar).

As noted in Section 3, the meaning of the measurements can depend on the method size. The measurement for two large methods might be the same as for two small methods, which would mean the large methods are much more similar than the two small methods, but the absolute values suggest they are equally similar. Also, the values of the counts can impact the measurement. The difference between two large counts (e.g. 100 versus 90) for a giving counting condition can be the same as for two small counts (11 versus 1), again indicating that the former is more similar than the latter, but just by the measurements they appear equally similar.

So some form of normalisation is needed. Unfortunately, the original paper does not describe how normalisation was performed, so we had to develop our own. We carried out a large number of trials of different forms of normalisation to find a form that have the best characteristics regarding false positives and false negatives (see below). We concluded the best normalisation was achieved by summing the values for the smaller count matrix, and dividing the raw difference measurement by that sum.

The false positive clone detection method mentioned in the original paper was also not described

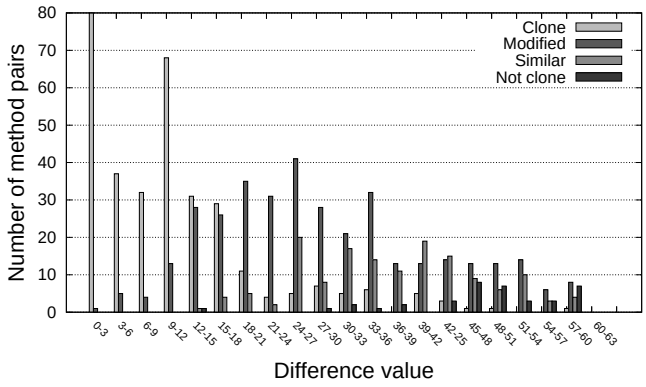


Figure 1: Showing the trade-off in candidate pair classification according to difference choices of threshold value. The 0–3 value has been truncated (from 179) for presentation purposes.

and thus we have come up with our own false positive detection method. As cloned fragments of code are similar or the same as the original fragment of code, we used a textual approach to discard clone pairs detected that had over 50% differences in text. This difference was computed after spaces and comments had been removed.

To improve performance, we classified method pairs by comparing the normalised difference between the two matrices to a predetermined threshold value. The threshold value was determined by analysing the distribution of difference values of clone pairs. This process consisted of:

1. Detecting all possible clone pairs for a selected software system and calculating the difference values.
2. Manually reviewing each method pair found and classifying it into one of the categories: clone, modified clone, similar but not clone, and not clone. (Also see below.)
3. Plotting a chart showing the distribution of different type of method pairs.
4. Determine the threshold value based on distribution.

Freecol version 0.8.0 was used for the analysis. The chart is shown in figure 1. From the chart, the default threshold value was chosen to be 45 to provide a balance between false positive and false negatives. A manual evaluation process like this was also used to evaluate different normalisation forms and choosing the level of text difference threshold.

Very small methods, such as getters and setters, are likely to look very similar. Also, even if they are created through cloning, identifying them as such is not very useful because they are so small. The original technique chose to ignore small methods for this reason, but did not specify how they identified such methods. In our implementation, the number of nodes in AST is used as the size of the method. A method is considered small if its size is less than a certain value. The number of lines of code was not used as the size of the method because it did not reflect the complexity of the code fragments and it may vary significantly depending on the coding style. By

looking at small methods such as getters and setters, we determined that an AST with 50 or fewer nodes could be reasonably classified as small.

Constructors are also ignored, as constructor clones are not very interesting. In addition, it is easy to get two constructors with the same variable count in a large system, and therefore they will introduce false positives.

### 4.2 Replication

We applied our implementation of the CMCD technique to the same 16 scenarios used by Yuan and Guo, and also to JDK 1.6.0_18. We did not have access to the student submissions and so did not replicate that part of their study.

### 4.3 Evaluation and Refinement

The original paper hinted that other counting conditions might be useful, so we planned from the beginning to support adding new conditions. That meant we also needed some way to select different conditions, and some way to show the results. We also needed to be able to vary various parameters, such as the choice of thresholds (see below). To support evaluation, we quickly learnt that it was important to not just see the list of candidate pairs, but to also show the contents of the pair, highlighting the differences between the two fragments. By examining candidate pairs in this way, we could then efficiently determine the accuracy of the choice of parameters by determining by manual inspection whether the candidate pair was indeed a clone pair. Finally, we needed the means to record the result of the manual inspection.

To this end, we developed a tool that can apply the foundation CMCD technique to any code base and that supports choosing different sets of counting conditions, different parameter values, reporting candidate clone pairs, highlighting the differences between a selected candidate pair, and recording the result of the manual inspection.

Candidate pairs are classified as "clone", "modified clone", "similar but not clone" or "not clone". "Clone" is where the method pair is clearly identical with minor changes such as differences variable types or variable names. "Modified Clone" is the same as "clone" but allowing a few addition or deletion of statements. "Similar but not clone" is used for classifying code clones where at a glance, they have lots of similarities in terms of structure and sub fragments of code, but is modified enough to not be considered clones. "Not clones" is where the method pair is clearly not a clone. This classification data can be saved for analysis and future clone detections, so that there is no need to reclassify clones when the detection process is rerun with different input parameters.

Clone pairs can be sorted based on any of the characteristics of the pairs (such as the value of the pre-normalised difference between a pair). This aids the identification of clone patterns in our results by ordering the results to allow for easy access to groups of data, and visualisation of correlation between data types. For example, clone pairs can be sorted by clone classification and then by the difference value of the method pair to determine if there is a correlation.

The tool we developed allows us to identify false positives (candidate pairs that are not in fact clones). Identifying false negatives (clone pairs that are never offered as candidates) is more challenging, however our tool also supports this because it allows us to easily change various parameters to the technique, in particular the thresholds. The choice of thresholds affects the level of false positives and false negatives — the higher the threshold the more false positives but the fewer false negatives. If we want to determine the degree of false negatives for a given threshold $t$, we can set the threshold to a value $l$ much larger than $t$, and examine those candidate pairs that are clones reported at level $l$ that are not reported at level $t$. These pairs are then false negatives at level $t$.

Identifying false negatives, as well as allowing us to provide error bounds on our results, also provides support for refining the technique. By examining false negatives, we can identify new counting conditions that may have the potential to detect such cases.

ant-1.8.0 argouml-0.34‡ c_jdbc-2.0.2 cayenne-3.0.1 cobertura-1.9.4.1 compiere-330 drawswf-1.2.9 freecol-0.10.3‡ freemind-0.9.0‡ ganttproject-2.0.9 gt2-2.7-M3 heritrix-1.14.4 hibernate-4.0.1‡ hsqldb-2.0.0 jFin_DateMath-R1.0.1 jag-6.1 javacc-5.0 jgraph-5.13.0.0‡ jgraphpad-5.10.0.2 jgrapht-0.8.1 jhotdraw-6.0.1‡ joggplayer-1.1.4s jrat-0.6 jrefactory-2.9.19 jruby-1.5.2 jtopen-7.1 marauroa-3.8.1 maven-3.0 nakedobjects-4.0.0 nekohtml-1.9.14 poi-3.6 pooka-3.0-080505 roller-4.0.1 sablecc-3.2 struts-2.2.1 sunflow-0.07.2 trove-2.1.0 velocity-1.6.4 wct-1.5.2 weka-3.6.6‡ xalan-2.7.1 xerces-2.10.0 xmojo-5.0.0

Figure 2: Systems used from Qualitas Corpus release 20120401. Systems for which multiple versions were analysed are indicated by ‡.

### 4.4 Empirical Study

Our empirical study was carried out on 43 open source Java systems from the Qualitas Corpus, release 20120401 (Tempero et al. 2010). We did both a breadth (different systems) and a longitudinal (multiple versions of the same system) study. The systems we used are listed in Figure 2, with those used for the longitudinal study marked by ‡. See the Qualitas Corpus website (`qualitascorpus.com`) for details of the systems studied, such has which files are analysed.

## 5 Results

In this section, we present the results of the different parts of our study. Their interpretation and consequences will be discussed in the next section.

### 5.1 Replication Study

As with the original CMCD implementation, our implementation was also successful at detecting clones for all 16 of Roy et al.'s scenarios. We also ran our implementation on the JDK 1.6 update 18 and found 11,391 similar methods in 2523 clone clusters. The process used 51 minutes using a 2.7GHz Intel Core i5 CPU.

### 5.2 Refinement

From the results of our replication study, we identified limitations in the original CMCD implementation. About 15% of the candidate pairs identified in our results were false positives. These false positives were recognised as clones mainly due to the choice of

$$normVCM = \frac{dVCM(total(vCM1) + total(vCM2))}{total(vCM1) + total(vCM2) + total(mCM1) + total(mCM2)}$$

Figure 3: Normalising difference between variable and method count matrices for two code fragments.

```
private Element remove(Connection connection,
                       Element element) {
  String address = connection.getSocket().
                   getInetAddress().getHostAddress();
  int port = Integer.parseInt(
                   element.getAttribute("port"));
  metaRegister.removeServer(address, port);
  return null;
}
```

Figure 4: Example of a method with no uses of variables according to the original counting conditions, and so has an empty count matrix (from the Freecol class `net.sf.freecol.metaserver.MetaServer`)

Table 2: The new Counting Conditions

| | |
|---|---|
| 11 | Variable used in first level while loop |
| 12 | Variable used in second level while loop |
| 13 | Variable used in third level while loop (or deeper) |
| 14 | Variable used in first level for loop |
| 15 | Variable used in second level for loop |
| 16 | Variable used in third level for loop (or deeper) |
| 17 | Variable used in switch-case statement |
| 18 | Method invoked |
| 19 | Method used in if-statement |
| 20 | Variable invoked on |

counting conditions. In Yuan and Guo's paper, the 13 counting conditions described were not sufficient to handle all the cases. For example, switch-case statements were ignored because the counts of variables did not reflect the existence of a switch-case statement.

Another issue was that code fragments that contained only method invocations had empty count matrices, despite potentially having non-trivial code. Figure 4 shows a small method with this property, but we saw a number of larger examples of this.

This lead us to change 3 existing conditions (11, 12, and 13 in Table 1) to represent the use of variables in loops at a more fined-grained manner, and added other conditions, including for method invocation. The new conditions are listed in Table 2.

As well as new counting conditions for variables, we also apply the same counting conditions to methods in a separate method count matrix. This matrix is normalised in the same manner as described for the existing (variable) count matrix as described in Section 4.1. With the two count matrices, the comparison of code fragments is done by determining the difference between the variable count matrices for each fragment and the method count matrices for each fragment. This again raises the issue that the respective sizes of the matrices could confound the result. For example, if the two variable count matrices are the same, but the method count matrices are different, then the size of the method count matrices might affect the result. Each pair of matrices is normalised and then the two normalised values are added together.

The normalisation function is shown in Figure 3. In that figure, *vCM1* and *vCM2* are the variable count matrices for code fragments 1 and 2 respectively, and *mCM1* and *mCM2* are the method count matrices; *total(*)* returns the sum of all values in the count matrix parameter; *dVCM* is the difference for the variable count matrices using the procedure described in Section 4.1; *normVCM* is the difference between the variable count matrices normalised with respect to the method count matrices.

### 5.3 RQ2: Empirical Study

We used our implementation on the systems listed in Figure 2 and the multiple versions of those systems indicated. In all, there were 310 different versions, involving 210,392 files and 26,702,561 non-comment non-blank lines of code (NCLOC). The total time taken was approximately 26 hours.

The results of the empirical study are summarised in Figure 5. The systems are ordered according to the number of methods we compared in each system (that is, ignoring "small" methods and constructors) in order to see if there are any trends due to some notion of system size. In fact were the systems ordered according to lines of code, the order would not be very different.

The figure shows three values: the *total* cloned code, that is, the percentage of the code (determined by non-comment non-blank lines of code — NCLOC) that appears in a clone cluster. The light grey shows the proportion of code that is *cloned* not counting the "original" that was cloned and the dark grey is the size of the *original* code that was cloned.

We show the total because we believe that that is what other studies report (although this is generally not stated) and we want to compare with them. However, we also believe that it is worthwhile seeing the size of the code that is cloned. If two systems have (for example) 10% total cloned code, but in one the original is 1% and the other it is 5%, then this difference is worth noting. Note that we do not really know which method was the original and which was cloned, but as they are all similar we can pick one as a representative (hence the use of quotes above).

The ranges of the values are: total 6.5% (`nekohtml`) – 59.5% (`cobertura`) with an average of 17% (`sunflow`), original 2.3% (`jrat`) – 11.8% (`cobertura`) with an average of 5.3% (`poi`), and cloned 3.8% (`nekohtml`) – 47.8% (`cobertura`) with an average of 11.7% (`pooka`). The medians are: total — 14.6%, original — 5.3%, and cloned — 10.0%, all by the system `poi`.

While there seems to be a slight trend of increasing cloned code with system size, the largest system (`gt2`) has 446,863 NCLOC and 13,174 methods, which is much bigger than the second largest, `jruby`, with 160,360 NCLOC and 7646 methods, and yet the amount of cloned code is less than many other systems (but see Section 6.2).

We examined the outliers, and found that a large proportion of generated code were included in these systems. Due to the nature of generated code, they could be similar or identical and therefore recognised

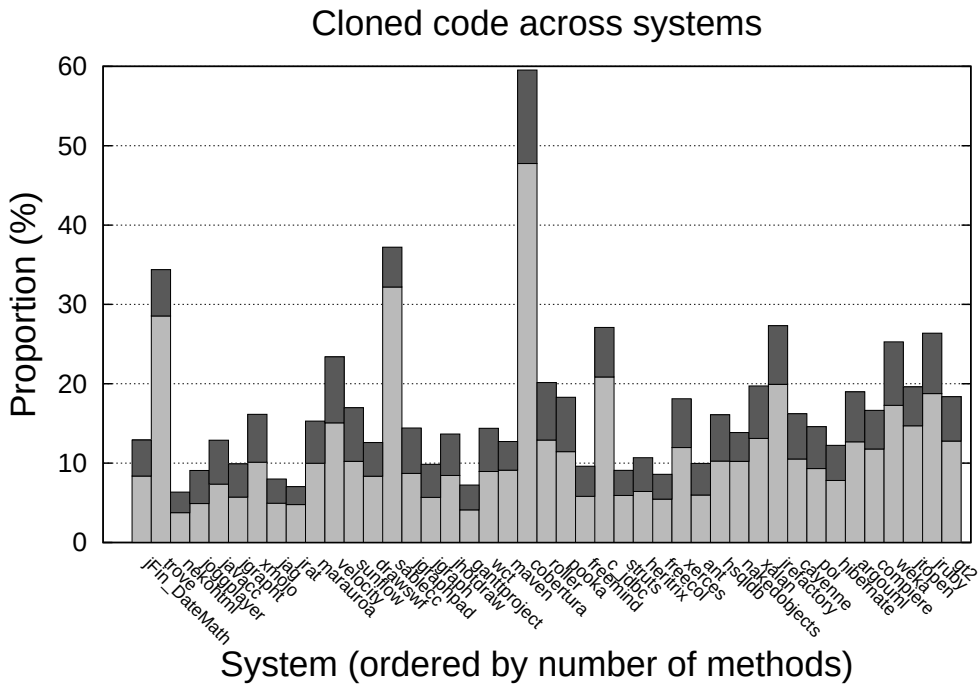


Figure 5: Proportion (%) of cloned code across the 43 systems in the study. The combined height of a bar is the proportion of code that appears in a clone cluster. The height of the dark grey bars shows the size of the "original" code that has been cloned.

as code clones by the clone detector. With the 3 top outliers (`trove`, `sablecc`, and `cobertura`) removed, the largest values are: total cloned 27.3%, original 8.3%, cloned 20.9%.

The process we used to determine the threshold value (the data is shown in Figure 1) also provides us with the means to estimate our false positive and false negative rates. For this study, we used a threshold value of 45. Those candidate clones with a difference value below the threshold (and thus reported by our tool as clones) that we manually classified as "similar" or "not clone" were classified as false positives, and those with a difference value above the threshold (that is, reported as not clones) but classified as "clone" or "modified clone" were classified as false negatives. Based on this, we had a false positive rate of 14.8% and false negative rate of 6.7%. We do note that all of the false positive method pairs found contained structurally similar code.

In addition to detecting clones in the latest version of the software systems in the Qualitas Corpus, different versions of software systems were also analysed. The results are shown in Figure 6.

## 6 Discussion

### 6.1 RQ1: Replication

The CPU time used between original implementation and our implementation was of the same order (our hardware is somewhat faster). However, the number of clones found was significantly different. Some of the clone pairs we detected were manually reviewed to assess the correctness of the result. A large proportion the clone pairs we found were the result of generated code. These generated code fragments were very similar to each other and therefore detected as code clones. We suspect that these methods were not considered in the original paper.

Yuan and Guo indicated that their implementation had a very low false positive rate, but did not provide any information on the false negative rate. Often there is a trade-off between false positives and false negatives, and so it is possible that their false negative rate was quite high. Since we had to develop our own normalisation and false positive elimination steps, it is possible that our false negative rate is not as high as the original. This might also explain why we found so many more candidate clone pairs.

Another possible source of variation was that it was not clear exactly which classes were examined in the original study, since Yuan and Guo analysed bytecode and we analysed source code.

While we did not get exactly the results reported by Yuan and Guo, they are close enough for us to conclude that the CMCD technique is as good as they claim. Furthermore, by manually reviewing detected clone pairs, there are clearly opportunities for improvement.

### 6.2 RQ2: Empirical Study

The smallest amount of cloned code we saw was 3.8% in nekohtml (6.5% if the original is included), which is the second smallest system (6,625 NCLOC and 185 methods) we analysed, meaning that the absolute amount of cloned code was also fairly small (421 NCLOC cloned code). Given that half of the systems we analysed (all larger than nekohtml) have 10% or more (14.6% if the original is included) points to non-trivial amounts of cloned code in open source Java systems. This is consistent with the findings of other studies.

Over the life-time of a system, according to Figure 6 there is again possibly a slight increasing trend over time for the systems that we have 20 or more versions for, however, as systems also grow over time, this might be further evidence of a relationship be-

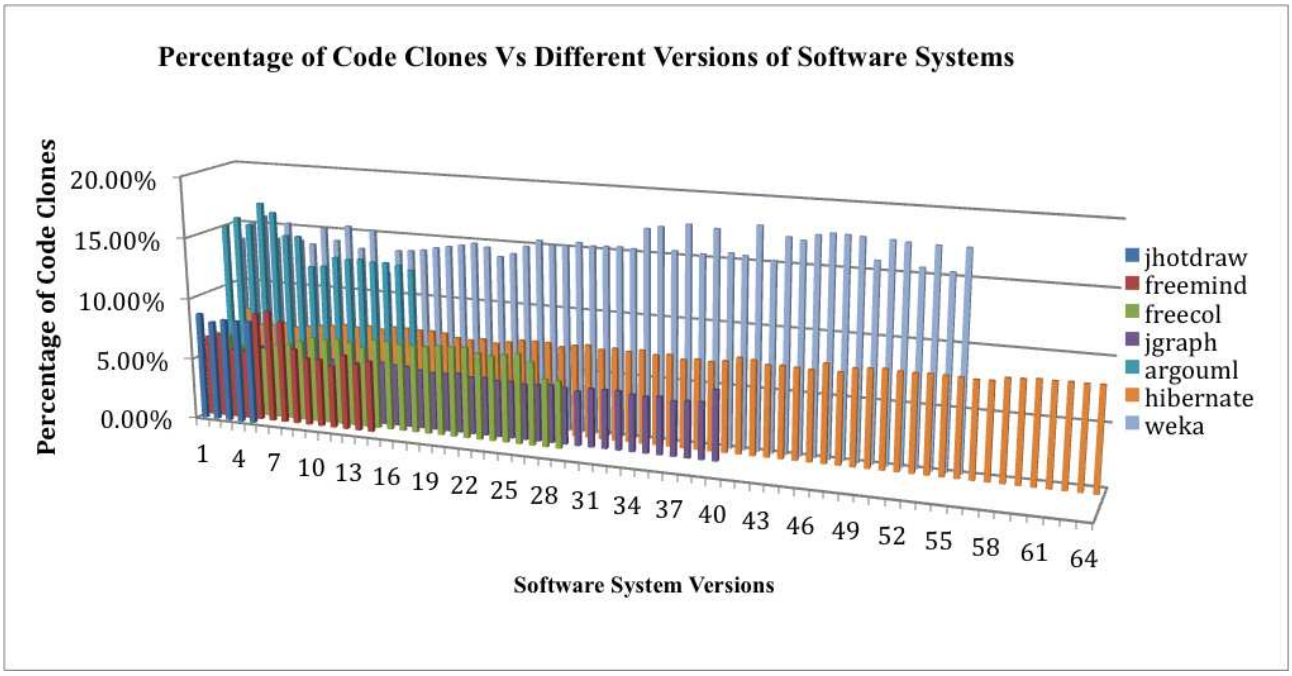


Figure 6: Study results showing the percentage of code clones across different versions of software systems

tween system size and amount of cloning. As we gather more data, we may be able to confirm this relationship.

It is worth noting that our implementation, like the original, has very good performance. We were able to analyse nearly 27 million NCLOC in about 26 hours on commodity hardware.

### 6.3 Threats to Validity

As with other clone detection research, a possible threat to the validity of our results is what we consider to be a clone. We have mitigated this threat by requiring that two people (the first two authors) agree on the designation (as described in Section 4) of each candidate pair.

Another possible threat is the correctness of our implementation. In particular, there is the possibility that some peculiar combination of circumstances will be mis-reported by our implementation. We have mitigated this through approximately 50 per-hours of manual review of candidate clone pairs.

One issue with comparing our results with others is that fact that we detect clones at the method level of granularity. This means that if the code in one method is completely copied to another method, but that other method also has at least as much code again added, we will not detect the use of cloning. We do not doubt that this happens, but our manual reviews found few such examples, leading us to conclude the impact on our results is small.

We have provided false positive and false negative rates for our results. These are based on our manual reviews, as supported by our tool, and so are necessarily a small subset of the total code base we analysed. While we cannot rule out missing incorrectly classified clone pairs, the nature of the CMCD technique is such that we believe our results are likely to apply generally.

Finally, we note that our results are generally in agreement with other studies, which gives us good confidence in them.

## 7 Conclusions

We have examined a technique for clone detection proposed by Yuan & Guo (2011) and found that generally their claims for its performance and accuracy are warranted. We have improved the original technique, in particular by adding more counting conditions and a separate method count matrix. Our improvements significantly reduce the false positives of the original. We confirmed the performance characteristics of the original study, being able to analyse nearly 27 million NCLOC in about 26 hours on commodity hardware.

We evaluate our improved CMCD through extensive manual validation supported by a visualisation tool. We replicated some of the original study and performed a large-scale empirical study of Java code. The study examined 43 systems in which we found that 1 in 2 systems had at least 10% cloned code, not counting the original. These results are broadly in agreement with other empirical studies. In particular, they are very close to a previous large study of systems written in C (Uchida et al. 2005). This suggests that the degree to which clones exist is not due to a particular language or style (procedural versus object-oriented).

In future work, we would like to improve the error bounds on the accuracy of our implementation and adapt it to work on sub-method granularity. While our empirical study is one of the largest performed for Java, it was not done on the whole of the Qualitas Corpus due to project constraints. We hope carry out an even larger study.

There is still much to be discovered about code clones. Based on our findings reported here, we believe the CMCD technique provides a very promising means to support such discovery.

## References

Baker, B. (1995), On finding duplication and near-duplication in large software systems, *in* '. . . , 1995., Proceedings of 2nd Working Conference on', p. 86.

Baker, B. S. (2007), 'Finding Clones with Dup: Analysis of an Experiment', *IEEE Transactions on Software Engineering* **33**(9), 608–621.

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

Bellon, S., Koschke, R., Antoniol, G., Krinke, J. & Merlo, E. (2007), 'Comparison and Evaluation of Clone Detection Tools', *IEEE Transactions on Software Engineering* **33**(9), 577–591.

Cartwright, N. (1991), 'Replicability, reproducibility, and robustness: Comments on Harry Collins', *History of Political Economy* .

Davey, N., Barson, P., Field, S. & Frank, R. (1995), 'The development of a software clone detector', *International Journal of Applied Software Technology* **3/4**(1), 219–236.

Drummond, C. (2009), Replicability is not reproducibility: nor is it good science, *in* 'Evaluation Methods for Machine Learning ICML Workshop', pp. 2005–2008.

Falke, R., Frenzel, P. & Koschke, R. (2008), 'Empirical evaluation of clone detection using syntax suffix trees', *Empirical Software Engineering* **13**(6), 601–643.

Johnson, J. (1994), Substring matching for clone detection and change tracking, *in* 'Proceedings International Conference on Software Maintenance ICSM-94', IEEE Comput. Soc. Press, pp. 120–126.

Juergens, E., Deissenboeck, F., Hummel, B. & Wagner, S. (2009), Do code clones matter?, *in* '31st International Conference on Software Engineering', IEEE, pp. 485–495.

Kamiya, T., Kusumoto, S. & Inoue, K. (2002), 'CCFinder: a multilinguistic token-based code clone detection system for large scale source code', *IEEE Transactions on Software Engineering* **28**(7), 654–670.

Komondoor, R. & Horwitz, S. (2001), Using slicing to identify duplication in source code, *in* '8th International Symposium on Static Analysis (SAS)', pp. 40–56.

Kontogiannis, K. (1997), Evaluation experiments on the detection of programming patterns using software metrics, *in* 'Proceedings of the Fourth Working Conference on Reverse Engineering', IEEE Comput. Soc, pp. 44–54.

Li, J. & Ernst, M. D. (2012), CBCD: Cloned buggy code detector, *in* '2012 34th International Conference on Software Engineering (ICSE)', IEEE, pp. 310–320.

Mayrand, J., Leblanc, C. & Merlo, E. (1996), Experiment on the automatic detection of function clones in a software system using metrics, *in* 'Proceedings of International Conference on Software Maintenance ICSM-96', IEEE, pp. 244–253.

Popper, K. (1968), *THE LOGIC or SCIENTIFIC DISCOVERY*, Routledge.

Roy, C. K., Cordy, J. R. & Koschke, R. (2009), 'Comparison and evaluation of code clone detection techniques and tools: A qualitative approach', *Science of Computer Programming* **74**(7), 470–495.

Schwarz, N., Lungu, M. & Robbes, R. (2012), On how often code is cloned across repositories, *in* '2012 34th International Conference on Software Engineering New Ideas and Emerging Results Track (ICSE)', Ieee, pp. 1289–1292.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H. & Noble, J. (2010), 'The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies', *2010 Asia Pacific Software Engineering Conference* pp. 336–345.

Uchida, S., Monden, A., Ohsugi, N., Kamiya, T., Matsumoto, K.-I. & Kudo, H. (2005), 'Software analysis by code clones in open source software', *Journal of Computer Information Systems* **45**(3), 1–11.

Vallée-Rai, R., Co, P. & Gagnon, E. (1999), Soot-a Java bytecode optimization framework, *in* 'Conference on the Centre for Advanced Studies on Collaborative Research (CASCON)', p. 13.

Yuan, Y. & Guo, Y. (2011), CMCD: Count Matrix Based Code Clone Detection, *in* '2011 18th Asia-Pacific Software Engineering Conference', IEEE, pp. 250–257.