Software Clone Detection and Refactoring

Francesca Arcelli Fontana*, Marco Zanoni*, Andrea Ranchetti*
and Davide Ranchetti*

*University of Milano-Bicocca, Viale Sarca, 336, 20126 Milano,
Italy, {arcelli,marco.zanoni}@disco.unimib.it,
{andrea.ranchetti,davide.ranchetti}@gmail.com

January 29, 2013

Abstract

Several studies have been proposed in the literature on software clones, from different points of view and covering many correlated features and areas, which are particularly relevant for software maintenance and evolution. In this paper we describe our experience on clone detection through three different tools, and our investigation on the impact of clone refactoring on different software quality metrics.

1 Introduction

In software programs we can find different kinds of redundancy or replication. Usually this kind of redundancy in the code is called *clone*; different definitions and taxonomies of clones have been proposed in the literature [8, 9]. With the term "clone" we often mean also "duplicate code"; duplicate code is an example of bad smell, as defined by Fowler [5]. The contributions of this experience report regard:

1. the experimentation of three different clone detection tools, PMD, Bauhaus and CodePro, on five versions of two open source software systems, Ant and GanttProject. We outline the difference in the detection results provided by the tools and we outline the evolution of clones in the five versions of the two systems. We observe if clones increase or decrease during the evolution of the system also with respect to an increment of the size of the system.

2. the analysis of the refactoring of code clones on the values of different software quality metrics. We considered metrics for cohesion, complexity and coupling. For the refactoring we decided to use the tool IntelliJ IDEA and for the metrics computation a plugin of IntelliJ IDEA, called Metrics Reloaded.

Through the first contribution, we aim to identify the principal differences of the three clone detection tools, possible advantages or drawbacks and outline

Table 1: Tools for duplicated code detection

Tool	**Supported Lang.**	**Algorithm**
PMD	Java, C, C++, Jsp, Php, Ruby, Fortran	Karp-Rabin's string matches [7]
Bauhaus	C, C++, C#, Java, Ada	Baxter's variation on AST [3]
CodePro	Java	Undocumented. Relies on Java AST.

some features on the results they provide, outlining the reasons of the differences in these results. We do not obviously aim to compare the results in terms of precision and recall, because a standard benchmark for clone detection is not yet available. As it has been outlined in previous work [1], the comparison of tools for code smell detection is very difficult and this is due, first of all, to the definitions of the smells, which are often ambiguous. Moreover, the detection techniques used by the tools are not always clearly stated, and the same holds for the thresholds of the metrics involved in the detection. As outlined by Kapser and Godfrey [6] also for clones, a widely accepted definition of clone is not yet available.

Through the second contribution, we aim to identify which are the software quality metrics most involved in the refactoring of code clones. Can we observe significant improvements in terms of software quality metrics through the refactoring of this smell? As already stated by Fowler [5] duplicate code smell represents probably the most critical one and hence the first one to be refactored. In the same time, as already suggested by Kasper [6], there are several situations where code duplication seems to be a reasonable or even a beneficial design option. Hence, it is correct and useful to detect clones in the code, but refactoring is not always desirable, and we could also face some risks in removing clones. In this paper we aim to investigate if we can gain some hints on the usefulness of removing clones, by analyzing the impact of clone refactoring on the value of particular software metrics. A first investigation of this kind, considering other smells, has been described in previous work [2].

The paper is organized through the following sections. In Section 2 we briefly introduce the clone detection tools we used. In Section 3 we report the results on clone detection through the three tools on five versions of two open source systems, and we outline the differences on the results and the evolution of the clones in the different versions of the systems. In Section 4 we analyze the impact of clone refactoring on different quality metrics. Finally, in Section 5 we conclude and outline some future developments.

2 Tools for Clone Detection

We briefly describe below the clone detection tools we exploited in our analysis.

PMD[1] scans Java source code and looks for potential problems like: duplicate code, possible bugs, and dead code. PMD allows the user to set the

[1] `http://www.PMD.sourceforge.net`

metrics thresholds for clone detection and allows to set the number of tokens of duplicated code; we chose to keep the default configuration (25 tokens).

Bauhaus[2] provides support to analyse and recover a system's software architecture; several maintenance tasks are supported as derivation of different views on the architecture of legacy systems, identification of re-usable components and estimation of change impact. The Bauhaus module for finding duplicated code looks for three type of clones: portions of identical code, their variation with different variable names and identifiers, and portions of identical code with added or removed statements.

Google CodePro Analytix[3] is a Java testing tool for Eclipse developers who are concerned about improving software quality. The main features are related to code analysis, metrics computation, JUnit test generation, dependency analysis and similar code analysis. For clone detection the tool offers three types of search: 1) *Code that can possibly be refactored*, 2) *Code that contains possible renaming errors*, 3) *Just looks similar*. We chose the last option to find as many duplicated code occurrences as possible.

In Table 1 we outline the supported languages and the algorithms used by the above tools. We chose these three tools because we had previous experiences in using them for software architecture reconstruction, code smell detection, dead code detection, and other program comprehension and assessment tasks. In this paper we describe our experience with these three tools, outlining the differences in the provided results. For a comparison and evaluation of code clone detection techniques and tools, please refer to the work from Roy et al. [10].

Other tools for duplicated code detection are for example CCFinder[4], IntelliJ IDEA[5], Checkstyle[6], CloneDr[7], Cpdetector[8], Jplug[9] and Code City[10].

3 Clone Detection and Evolution

We now describe our analysis on the detection of clones on the GanttProject[11] and Ant[12] systems. During this analysis we often use the term *occurrence*; an occurrence is a portion of code that is duplicated at least one time. A large debate about the definition of clones can be found in the literature. As we will observe in this section, each clone detection tool adopts its own definition.

As mentioned in the previous section, two pieces of code do not have to be identical to be considered clones, but they can have also little differences. According to the well known classification of clone types [10], we have:

- Type I: Code fragments are identical except for variations in white space, layout, and comments.

[2] http://www.bauhaus-stuttgart.de
[3] https://developers.google.com/java-dev-tools/codepro/doc/
[4] http://www.ccfinder.net
[5] http://www.jetbrains.com/idea
[6] http://checkstyle.sourceforge.net
[7] http://www.semdesigns.com/Products/Clone/
[8] http://cpdetector.sourceforge.net/
[9] http://jplug.sourceforge.net/
[10] http://www.inf.usi.ch/phd/wettel/codecity.html
[11] http://www.ganttproject.biz
[12] http://ant.apache.org/

Table 2: Ant versions

Version	Released on	# classes	# methods	LOC
1.5.2	2003	950	6761	70205
1.6.1	2004	1267	9779	98882
1.6.5	2005	1341	10510	105117
1.7.1	2008	1592	12496	117474
1.8.2	2010	1742	13424	128253

Table 3: Ant results: PMD

Version	occurr.	duplicated LOC	% duplicated code
1.5.2	876	10930	15.56%
1.6.1	1214	15557	15.73%
1.6.5	1221	16185	15.39%
1.7.1	1244	17277	14.70%
1.8.2	1415	20048	15.63%

- Type II: Code fragments are structurally and syntactically identical except for variations in identifiers, literals, types, layout and comments.
- Type III: Code fragments are copies with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.
- Type IV: Two or more code fragments perform the same computation but are implemented through different syntactic variants.

Respect to this classification of clone, Bauhaus and CodePro find clones of Type I, II and III; PMD instead finds clones of Type I and II.

3.1 Clone Detection in Ant

In Table 2 we report a summary of the main features of the five analysed versions of Ant. Our first analysis of Ant was done with PMD. This tool was very fast in the code duplication search. PMD considers an occurrence of duplicated code a portion of code that is duplicated at least one time and that is at least composed by 25 tokens. One line of code is composed by several tokens, depending on the programming style; in the analysed system 25 tokens correspond to 4–6 LOC. We report a summary of the results in Table 3. From these results we can observe that the number of duplicated lines of code grew for each new version. The percentage of duplicated code in the system is more or less constant in all the analysed versions, meaning that duplicated code grows at the same rate of the size of the system.

Bauhaus results are different from those of PMD. From version 1.5.2 to 1.7.1, both duplicated LOC and percentage of duplicated code grew a lot. In version 1.8.2, instead, both values decreased. This result suggests that some refactoring effort was spent between versions 1.7.1 and 1.8.2. To understand why the results of PMD and Bauhaus are so different we point to the discussion about the

Table 4: Ant results: Bauhaus

Version	occurr.	duplicated LOC	% duplicated code
1.5.2	135	7352	10.47%
1.6.1	214	14548	14.71%
1.6.5	294	20952	19.93%
1.7.1	479	28748	24.47%
1.8.2	296	24087	18.78%

Table 5: Ant results: CodePro Analytix

Version	occurr.	occurr./LOC
1.5.2	516	0.73%
1.6.1	669	0.67%
1.6.5	708	0.67%
1.7.1	717	0.62%
1.8.2	805	0.61%

relationships between the number of occurrences and their size, reported in Subsection 3.1.

The analysis performed with CodePro was done with the most exhaustive search method available. Unfortunately, CodePro does not provide the number of duplicated lines of code. To compare the results of CodePro with those of PMD and Bauhaus, we consider the number of occurrences and the occurrences/LOC ratio, where LOC is the number of lines of code of the entire system. The purpose of the ratio is to balance the duplicated code size with the size of the system. We can see that even if the occurrences increased in every version, the occurences/LOC ratio decreased constantly, suggesting that in every new version the developers introduced less duplicated code than in the previous one.

Considerations on the results PMD and CodePro agree on the fact that the number of occurrences increased in each new version, and in particular in versions 1.6.1 and 2.0.9. Bauhaus instead shows that the occurrences decreased significantly in the last version. Bauhaus found less occurrences than the other two tools. The reason of this result is that Bauhaus finds larger occurrences, because it exploits less restrictive rules.

In Figure 2 we can see a comparison of the three tools based on the occurences/LOC parameter. Figure 1 and Figure 2 are similar: In both of them PMD scores the highest values and Bauhaus the lowest, and even the shape of the lines is similar in both figures. The difference is that in Figure 1 occurrences increase slightly in each new version (except for the latest version in which there is a decrease of Bauhaus), while in Figure 2 the parameter has a decreasing trend (with Bauhaus as an exception also this time).

3.2 Clone Detection in GanttProject

In Table 6 we report a summary of the main features of the five versions of GanttProject we analyzed. These versions cover five years of development,

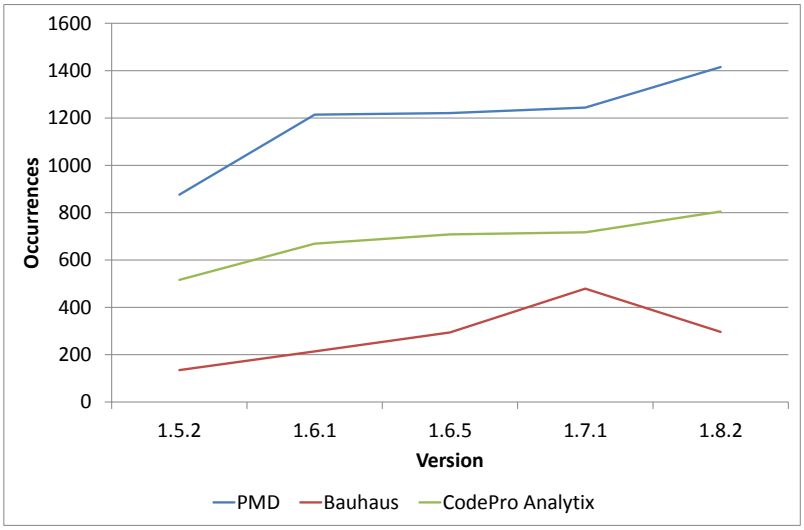

Figure 1: Occurrences in Ant

Table 6: GanttProject overview

Version	released in	# classes	# methods	LOC
1.10.2	2004	396	1924	21219
1.11.1	2005	549	2724	26337
2.0.0	2006	962	4931	44399
2.0.6	2008	1062	5450	47767
2.0.9	2009	1076	5530	48595

during which the system grew over two times.

Looking at the results of PMD in Table 7 we can observe that, with the development of version 1.11.1, both the percentage and the size of duplicated line of code decreased significantly. In version 2.0.0 the lines of code of the project and also the duplicated lines of code increased a lot. From version 2.0.0 to 2.0.9 instead, duplicated lines of code stabilized around five thousand LOC and the percentage of duplicated code continued to decrease to almost 10%. Considering the results for Bauhaus in Table 8, we can say that even if duplicated LOC are quite different from those of PMD, the trend of the percentage of duplicated code is very similar.

Looking at Table 7 we can see that the trend of occurrences/LOC for PMD is similar to the one of the percentage of duplicated code for Bauhaus and CodePro shown in Table 8 and Table 9. The three tools agree in saying that the version containing more duplicated code is 1.10.2.

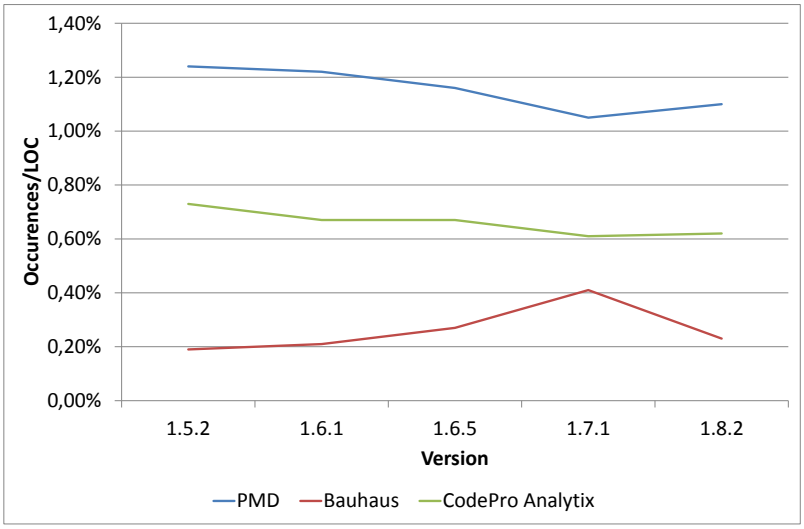

Figure 2: Ant occurrences/LOC comparison

Table 7: GanttProject results: PMD

Version	occurr.	duplicated LOC	% duplicated code
1.10.2	347	3159	14.88%
1.11.1	316	2495	9.47%
2.0.0	499	5156	11.61%
2.0.6	677	5102	10.68%
2.0.9	692	5073	10.43%

Considerations on the results In Figures 3 and 4 we show some of the results that we consider most significant.

Looking at Figure 3 we can see that, excluding the second version, occurrences of duplicated code always increased. PMD and CodePro got similar results: found occurrences were between 200 and 800. Bauhaus instead, as already observed, always finds fewer occurrences. Observing instead the duplicated lines of code, it emerges that PMD and Bauhaus results are very similar. The first tool finds more occurrences, but smaller, the second one fewer occurrences, but bigger. The result is that the duplicated lines of code found by the two tools are very similar.

All the tools show that version 1.11.1 had a significant reduction of duplicated code. The next version instead brings back the percentage to high levels. The last two considered versions reduce it slightly. In Figure 4 we can see a comparison among all the considered tools. As in the Ant analysis, we used the occurrences/LOC parameter. In this case the three tools trends are identical; after a significant reduction, the occurrences of duplicated code respect to the

Table 8: GanttProject results: Bauhaus

Version	occurr.	duplicated LOC	% duplicated code
1.10.2	54	1808	8.52%
1.11.1	37	1455	5.52%
2.0.0	54	3568	8.03%
2.0.6	55	3497	7.32%
2.0.9	53	3311	6.81%

Table 9: GanttProject results: CodePro

Version	occurr.	occurr./LOC
1.10.2	232	1.09%
1.11.1	242	0.91%
2.0.0	360	0.81%
2.0.6	379	0.79%
2.0.9	395	0.81%

project size increased a little and then stabilized. This is a confirmation of the validity of the tools and of this comparison. Every tool agree on the fact that version 2.0.0 is the less affected by duplicated code.

3.3 Considerations on the tools

The main differences lie in the number of occurrences found by the tools and the size of these occurrences. These differences are due to the different algorithms used to detect duplicated code. In fact we notices that Bauhaus allows little differences in the clones, resulting in larger detected occurrences. On the contrary, PMD find a lot of small occurrences, that are identical clones, ignoring slightly different clones. Considering these results and the experience we had with these three tools we found Bauhaus to be very useful, because it provides ordered and easy enumerable results, and allows users to choose among many search option. PMD provides clear and ordered results, but it allows only to set the number of tokens and to find only identical duplicated code; CodePro instead, as Bauhaus, has more search options, but provides results which are difficult to count and analyze. Through Bauhaus we are able to find fewer occurrences of large size respect to a lot of occurrences of small size. We think that this aspect is important because, as we observe in Section 4, we found more useful to refactor less large duplicated code blocks than many small ones.

4 Impact of refactoring on system quality

In this section we analyse the impact of clone refactoring on system quality through the evaluation of different metrics values. The considered systems are Ant 1.8.2 and GanttProject 2.0.9.

The metrics we evaluated are reported in Table 10. They are calculated

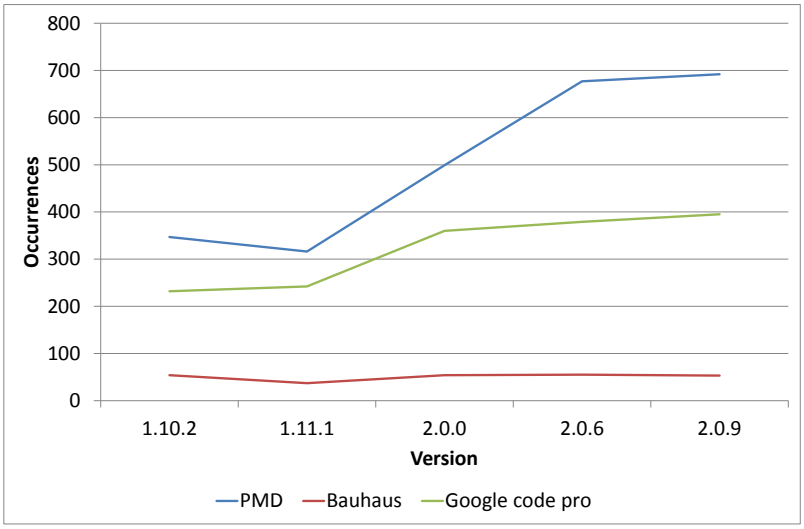

Figure 3: Occurrences in GanttProject

at package or class level. We used the Metrics Reloaded[13] plugin for IntelliJ IDEA[14] to compute the metrics values. We chose the same IDE for refactoring, as it provides many features for code analysis.

We applied some of the widely adopted refactoring techniques for duplicated code proposed by Fowler [5]:

- Extract Class: create a new class and move the relevant fields and methods from the old class into the new class. When there are two classes with a very similar or identical subclass extracting these classes in one class avoids duplicated code.
- Extract Method: there two or more code fragment in different classes that can be grouped together.
- Replace Method: allows to find code repetitions similar to the selected method and replace them with calls to the method.

To evaluate the impact of refactoring on the quality metrics of the analysed system we organized our work in the following steps:

- we evaluated the 8 metrics on the analyzed systems before any refactoring;
- we chose which classes and methods to refactor by observing the results provided by the three described tools for clone detection;

[13] `http://plugins.intellij.net/plugin/?id=93`
[14] `http://www.jetbrains.com/idea`

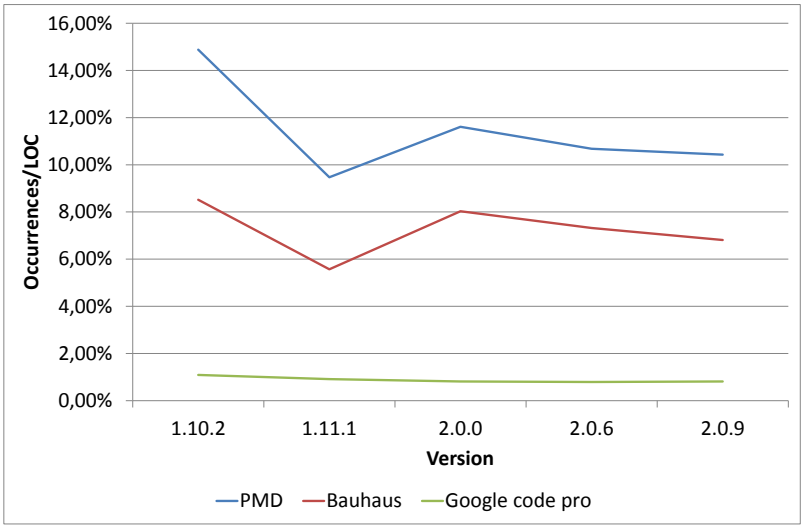

Figure 4: GanttProject occurr./LOC comparison

- we computed again all the metrics values after applying the refactoring;
- we evaluated the impact of refactoring by comparing the values of the metrics before and after the refactoring.

4.1 Ant Refactoring

The metric values shown in Table 11 refer to the `optional` package of Ant 1.8.2. The package is composed of 340 classes. The values for the metrics at class level, instead are shown in Table 12. A minus sign (−) means a negative impact on the metric value, and a plus sign (+) a positive impact; an equal sign (=) means that the value did not change. For example, if the complexity decreases, this represents a positive impact, and hence we have a plus sign (+) for CC or WMC metrics.

We can observe that refactoring operations reduced the lines of code. In fact, as we can see in Table 11 the LOC value decreased by 0.38%. The number of methods instead decreases only if the refactoring operations delete more methods than the created ones: the Extract Method refactor technique creates a method, while the Replace Method technique deletes a method. In this case the NOM value decreased by 0.65%. This result is justified by the fact that, in most cases, the refactoring consisted in extracting classes and methods to be used by multiple clients; this is also the reason why also the CF value decreased. The CC metric value decreased by 0.65%; in fact, the decomposition of functions, realized by the refactoring technique Replace Method, is a well known method for reducing the Cyclomatic Complexity. Refactoring duplicated code improves also the stability of the project and the value of abstractness: the DMS value

Table 10: Metrics

Metric	Description	Level
LOC	Line Of Code	Package
NOM	Number Of Methods	Package
CC	Cyclomatic Complexity	Package
CF	Coupling Factor	Package
DMS	Distance from the main sequence	Package
WMC	Weighted method complexity	Class
LCOM	Lack of cohesion in methods	Class
RFC	Response for a Class	Class

Table 11: Impact of refactoring at package level

Version	LOC	NOM	CC	CF	DMS
Ant 1.8.2	+0.38%	+0.65%	+0.65%	+0.14%	+25%
GanttProject 2.0.9	+0.25%	=	+0.18%	=	−3,84%

decreased by 25%. The results in Table 12 show that the metric values of the refactored classes improved considerably. Through the Extract Method and the Replace Method refactorings, the WMC value decreased significantly. RFC decreased with WMC, because it is well known that a correlation exists between the two metrics. LCOM value decreased because, with respect to the initial state of the system, the application of the Replace Method refactoring resulted in fewer methods accessing the same attributes of the refactored class. Extract Method and Extract Class techniques influenced positively LCOM: it is well known that these techniques improve cohesion between methods and classes [4].

4.2 GanttProject refactoring

The metric values shown in Table 11 refer to the `ganttproject` package of GanttProject 2.0.9. The considered package is composed of 983 classes. The metric values at class level are shown in Table 12.

The values of the metrics at the package level have not undergone many changes. The reason is that the occurrences of duplicated code in GanttProject are, in many cases, not refactorable. Some instances of duplicate code, as getter and setter methods, cannot be refactored. These observations are supported by the fact that the average size of occurrences of Ant is higher than that of GanttProject. In fact, Bauhaus reports that the average size of occurrences in the version 1.8.2 of Ant is 81.4 lines, and in the version 2.0.9 of GanttProject is 62.5 lines.

The number of methods remained unchanged because we used the same number of times Extract Method as Replace Method and Extract Class. In fact, Extract Class and Replace Method eliminate methods, while Extract Method adds them.

Table 12: Impact of refactoring at class level

Version	LCOM	WMC	RFC
Ant 1.8.2	+32.3%	+8.41%	+1.78%
GanttProject 2.0.9	+6.32%	+1.34%	+14.06%

In Table 12 we see that the value of RFC changed a lot. The reason is that the consequence of reducing the code complexity by deleting classes and methods is that the number of external calls has been reduced. Furthermore, the new methods do not introduce complexity because they are composed by existing code. The values of the class metrics changed more than the package ones, because many of the refactored classes are small; so they have a small impact on the package. GanttProject is composed by many small classes, some of which are highly coupled and contain duplicated code.

Considerations on the results Looking at the data collected so far, we can make some observations. The results suggest that there is a correlation between the size of an occurrence of duplicated code and the chances that this occurrence can be refactored. Looking at the Bauhaus results in Table 4 and Table 8 we can see that in Ant 1.8.2 the average size of an occurrence is 81.4 lines, instead in GanttProject 2.0.9 the average size of an occurrence is 62.5 lines. A direct consequence of these results is that refactoring in Ant was much more significant and this is evident by observing the results in Table 11 and Table 12. It is also clear that not every single occurrence can be refactored; in fact, many of them are too complex or simply not refactorable. For example, many instances of duplicated code are portion of cycles or conditionals that whose extraction is not convenient. Another example of hard-to-refactor clone is when two classes in two different packages have a similar method; in this case the Extract Class technique may cause more problems than it solves. A good way to reduce the number of false positives during clones search is to increase the number of tokens. We already observed that metrics at class level changed more that the ones at package level. The metrics confirm the intuitive conjecture that refactoring is useful to improve the quality of a specific class, in particular if this class is complex and highly coupled with other classes, but working a lot on simpler classes does not improve project quality significantly.

We used the most known and verified refactoring techniques, but it is possible we did not removed every occurrence that could be corrected; in any case, we refactored most of the found occurrences.

4.3 Not refactorable clones

As we said before, not all clones found by the tools are refactorable. When a clone is of type I and II it is simple to remove; in most cases using extract method is the right solution. When instead two clones are similar, but with some different statements, maybe these two pieces of code perform different tasks; in this case, if it is possible, we can extract the common part and create a new method.

An example of refactoring (Extract Method) that is not possible is when two clones, assumed identical by the tool, belong to two different classes and packages and use class parameters. In the code fragment reported below we see an identical clone found in classes `CustomColumnsPanel` and `GanttTreeTable`, which are in different packages.

```
.undoableEdit("PopUpNewColumn",
  new Runnable() {public void run() {
      CustomColumn customColumn
        = new CustomColumn();
      GanttDialogCustomColumn d
        = new GanttDialogCustomColumn(
            myUIfacade, customColumn);
      d.setVisible(true);
      if (d.isOk()) {
```

In cases like this, refactoring brings more disadvantages than advantages, so it is convenient not to modify the code. Extracting classes that have to be shared among packages increases dependencies between packages and do not decrease significantly LOC in the case in which there are only few duplicated lines.

Here we present an other example of clones. This first piece of code is from `ProjectSettingsPanel.java`:

```
JButton bWeb=new TestGanttRolloverButton(
        new ImageIcon(getClass().getResource(
        "/icons/web_16.gif")));
bWeb.setToolTipText(GanttProject.getToolTip(
        language.getText("openWebLink")));
bWeb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
```

This second piece of code is similar, but with some differences; it is from `GanttTaskPropertiesBean.java`:

```
JButton bdate = new TestGanttRolloverButton(
        new ImageIcon(getClass().getResource(
        "/icons/clock_16.gif")));
bdate.setToolTipText(GanttProject.getToolTip(
        language.getText("putDate")));
bdate.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
```

It is not convenient to refactor these couple of clones because they use private class parameters and only few characteristics are similar. So creating a new service method to execute only these few simple line of codes is not worth the effort and can introduce new problems.

Concluding this section we can say that to reduce these false positive is better to look for larger occurrences, e.g., increasing the number of tokens in PMD and Bauhaus. CodePro instead, does not allow to set the size of occurrences.

4.3.1 Threats to validity

Related to the threats to validity for the tools of clone detection, we choose tools we know and used several times and as we observed a comparison is diffcult as the validation of their results. We check for the existence of the clones we detected and on which we applied the refactoring steps. The number of tools considered is not high, but the three tools use quite different algorithms and search criteria, so we think that a preliminary comparison, in particular when they get similar results, is significant. Related to the refactoring validity we use the most known and verified techniques, as *extract method*, *replace method* and *extract class*.

Related to the consideration on results, we performed our analysis individually and later we compare and discuss them in order to consider only the most consolidated results.

5 Conclusions and Future Developments

In this paper we described our experience on clone detection through three different tools and analyzing five version of two opens source systems.

The tools exploit different detection techniques and hence provide different results, and comparing these results is not an easy task because of the different sizes of the reported occurrences and the different clone types. For example, there is a general tendency to keep the percentage of duplicated code stable among releases of the same project, but this is not confirmed by all the tools (Bauhaus outcomes are less stable than the others).

We also manually removed, by refactoring, the duplicated code occurrences in every case that we found possible and reasonable, without directly introducing bad design practices. The analysis of a set of quality metrics before and after the refactoring gave confirmation on the fact that removing duplicated code leads to an improvement in most cases, and that it should be focused on the more complex and larger classes. In particular, we observed an improvement on the stability of the system (25% of DMS for Ant), an improvement in the cohesion (32% of LCOM for Ant), a decrement in coupling (14% RFC for GanttProject), a decrement in complexity (8% WMC for Ant) and obviously a reduction in the LOC. Moreover, we observed that the metrics at class level improve more than those at package level, hence refactoring is useful to improve the quality of specific classes.

In future work, we would like to investigate if the refactoring of clones can lead to the removal or introduction of new code smells. Hence the plan is to detect several other smells on the system we have analysed in this paper and check their presence before and after the refactoring of the clones. We are interested also in analyzing the impact on software quality by considering different refactoring choices.

References

[1] F. Arcelli Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5:1–38, aug 2012.

[2] F. Arcelli Fontana and S. Spinelli. Impact of refactoring on quality code evaluation. In *Proceedings of the 4th Workshop on Refactoring Tools (WRT '11)*, pages 37–40, New York, NY, USA, 2011. ACM.

[3] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE Computer Society, November 1998.

[4] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring - improving coupling and cohesion of existing code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 144–151. IEEE Computer Society, November 2004.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1999. `http://www.refactoring.com/`.

[6] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pages 19–28, October 2006.

[7] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[8] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007.

[9] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, School of Computing, Queen's University, Kingston, Ontario, Canada, September 2007.

[10] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009. Special Issue on Program Comprehension (ICPC 2008).