Archeology of Code Duplication : Recovering Duplication Chains From Small Duplication Fragments

Richard Wettel Radu Marinescu

LOOSE Research Group
Institute e-Austria Timişoara, Romania
{wettel,radum}@cs.utt.ro

Abstract

Code duplication is a common problem, and a well-known sign of bad design. As a result of that, in the last decade, the issue of detecting code duplication led to various solutions and tools that can automatically find duplicated blocks of code. However, duplicated fragments rarely remain identical after they are copied; they are oftentimes modified here and there. This adaptation usually "scatters" the duplicated code block into a large amount of small "islands" of duplication, which detected and analyzed separately hide the real magnitude and impact of the duplicated block. In this paper we propose a novel, automated approach for recovering duplication blocks, by composing small isolated fragments of duplication into larger and more relevant duplication chains. We validate both the efficiency and the scalability of the approach by applying it on several well known open-source case-studies and discussing some relevant findings. By recovering such duplication chains, the maintenance engineer is provided with additional cases of duplication that can lead to relevant refactorings, and which are usually missed by other detection methods.

Keywords: code duplication, design flaws, quality assurance

1 Introduction

Duplicating code, while easy and cheap during the development phase, moves the burden towards the already overloaded and much more expensive maintenance phase. Fowler and Beck ranks it first in their list of "bad smells in code" [6] and we strongly believe they were right. Therefore, we will not emphasize the consequences of introducing duplicated code.

In a line-based approach, large blocks of code affected by modifications (renaming of variables or even statement insertions or removals) would be identified as small, less important fragments of duplicated code, apparently not related to each other.

To address this issue, we propose an approach that can merge such small fragments that belong together and provides the maintainer with some additional duplication blocks otherwise granted with less importance.

2 The Archeology Metaphor

Like an archeologist who puts together all the ruins of an ancient village in order to build a complete picture, rather than analyzing each artifact separately, we try to recover a close representation of a scattered duplicated block, in order to make the right refactoring decisions.

2.1 It Started with a Scatter-Plot

The scatter-plot approach, successfully applied in the code duplication detection field starting with the early '90 ([1], [4], [5]), uses a visual representation that can point out "dark" areas, which possibly host problems. Our approach provides the reengineer with results in form of a list of duplication chains. However, since the visual idea of a scatter-plot is behind our detection algorithm, we chose it as a means to illustrate the various concepts throughout this paper.

2.2 Need for Duplication Chains

Imagine we have the two pieces of Java code from Figure 1. Despite the fact that it seems obvious that they have common origins, due to the deleted and modified lines of code, they could be detected as 3 smaller clones, which is rather false. In a more pessimistic scenario, they would be filtered out by the minimum length threshold. One could rightly argue that there are approaches which can detect variables renaming. What if the lines of code are modified further than just variables or if there are lines appearing in only one of the two code fragments?

```
initSensors(tSensors);
readSensors(tSensors);
lcd.init();
int i = 0;
while(i < tSensors.length){
  temp[i] =tSensors[i].getTemp();
  lcd.println("T"+i+"="+temp[i]);
  i++;
}
regulateTemp(temp);
```

```
initSensors(tSensors);
readSensors(tSensors);
int i = 0;
while(i < tSensor.length){
  temp[i] =tSensor[i].getTemp();
  System.out.println("T"+i+"="+temp[i]);
  i++;
}
regulateTemp(temp);
```

Figure 1. Scattered duplication

Moreover, detected clones might not be relevant if they are too small or analyzed in isolation. Our main goal is to capture, along with the usual clones, blocks of scattered clones that may have common origin, which we will further refer to as **duplication chains**.

2.3 Anatomy of a Duplication Chain

A *duplication chain* can be a complex element (the representation of the recovered duplicated code block), composed of a number of smaller, exact clones (further referred as exact chunks), separated pairwise by non-matching gaps. Figure 2 illustrates the previous example's scatter-plot representation, where the marked cells correspond to the matching pair of lines of code intersecting in that precise point.

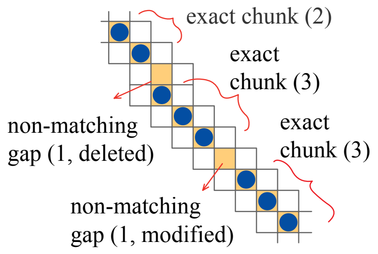


Figure 2. Duplication chain

An *exact chunk*, put in the context of the archeology metaphor, is a non-altered part of a duplicated block, that preserved its identity. An exact chunk appears in a scatter-plot as a continuous diagonal, as it can be seen on Figure 2.

A *non-matching gap* reflects the changes that have been made to the originating duplicated block, in terms of lines of code (insertion, deletion, modification). Thus, while apparently less important in clone detection, these non-matching parts provide us with extra information about the adaptation process. In a scatter-plot representation, non-matching gaps appear as shortest non-marked paths linking two consecutive diagonals (Figure 2).

A characteristic of a duplication chain, directly related to the adaptation process is the *signature*, which captures the structural configuration in terms of exact chunks, non-matching gaps and the metrics around them. In terms of the archeology metaphor, the signature could be associated with a "map" storing the places where all the related items where discovered. The signature of the previous example is "E2.D1.E3.M1.E3", which describes two code fragments having 3 exact (E) chunks of sizes 2, 3 and 3, separated by 2 non-matching gaps: one with 1 deleted (D) line and the other with 1 modified (M) line of code.

2.4 Proportional Harmony

In the context of size, we want to capture only those code fragments pairs that contain a *significant* amount of duplication. While an exact clone is significant if the clone's size is larger than a threshold, a significant duplication chain must also be proportionally harmonious. First, we will define some metrics related to these proportions (measured in LOC).

Size of Exact Chunk (SEC) reflects the degree of the granulation left behind by the adaptation phase of the copy-paste-adaptation process. *Line Bias* (LB) is the size of non-matching gaps and its value may allow us to decide if two exact chunks belong to the same duplicated block, since it provides a measure of distance between them. *Size of Duplication Chain* (SDC) is the size of the more meaningful block of duplication, which actually suggests its magnitude.

In order to constrain the duplication chain's proportions, we will set a minimum SDC to filter the less relevant clones. Furthermore, we will impose a minimum SEC and a maximum LB. In the harmony context, there is a relation between SEC and LB: the SEC should always be larger than LB, because it is not desirable to detect duplication chains with gaps larger than its exact chunks.

2.5 Stepwise Recovery Methodology

We propose an approach of lightweight line-matching, enhanced with the concept of chain duplication, which can also cover duplications that cannot be detected by a simple line-matching approach.

Phase 1: Code Preprocessing. After reading the source-files, we eliminate the white spaces, so that the various indentation styles would not make the difference. An optional feature is the possibility to ignore the comments in the analysis process. This phase provides a set of relevant (clean) lines of code.

Phase 2: Populate the scatter-plot. As in the original scatter-plot approach, we compare every line of code with every line of code in the project. As a result of this comparison, the matrix will be divided in two symmetric areas, around the main diagonal, which is always completely marked (self comparison). We then populate only one half of the matrix, in order to avoid storing redundant information. The matching intersections are marked.

Phase 3: Build the duplication chains. Starting with the left-upper matrix cell, we look for the first marked one, as a starting point for a possible duplication chain. From here, we accumulate the marked cells following the diagonal direction towards the lower-right cell. The algorithm will accept as an extension of the chain either a marked cell that continues an exact chunk or a marked cell situated in its vicinity, whose range is controlled by the maximum LB. Significant duplication chains will be stored in a results list.

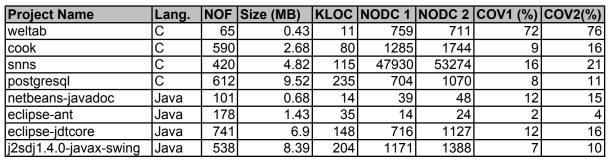
3 Validating the Approach

In order to present the advantages over traditional code duplication detectors, we have to prove that this approach provides additional relevant duplications, usually missed by other line-based detection methods. To demonstrate this, we applied the proposed approach over a set of case studies. DuDe, our supporting tool owes its flexibility to the tunable thresholds which can filter the results based on size and proportional harmony. The tool provides a list of suspects which can be further analyzed, by means of the duplicated code visualization feature and statistical information.

3.1 Quantitative Gain

We took 8 Java and C projects, covering the size range from 0.5 MB to 10 MB, containing between 11,000 and 235,000 LOC in a number of 65 to 741 files. We compared traditional approach results (NODC1, COV1) with the one based on duplication chains (NODC2, COV2). Correlating the results in terms of *coverage*[1] and number of duplication chains presented in Figure 3, we can state that our enhanced "archeological" approach provides an important amount of otherwise lost code duplication information.

[1]Coverage is the ratio of the number of copied lines of code to the total number of lines in the system



Project Name	Lang.	NOF	Size (MB)	KLOC	NODC 1	NODC 2	COV1 (%)	COV2(%)
weltab	C	65	0.43	11	759	711	72	76
cook	C	590	2.68	80	1285	1744	9	16
snns	C	420	4.82	115	47930	53274	16	21
postgresql	C	612	9.52	235	704	1070	8	11
netbeans-javadoc	Java	101	0.68	14	39	48	12	15
eclipse-ant	Java	178	1.43	35	14	24	2	4
eclipse-jdtcore	Java	741	6.9	148	716	1127	12	16
j2sdj1.4.0-javax-swing	Java	538	8.39	204	1171	1388	7	10

Figure 3. Experimental results

3.2 Quality-Focused Analysis

In order to validate the quality of our results, we extracted the clones found in another project (JHotDraw) only by the duplication chain approach and analyzed them manually. Out of 72 clones, there were 30 duplication chains. Summarized, we found over 76% relevant clones, potential subjects to refactorings.

In order to calculate the *recall*[2] of our tool, we considered only the type 1 (exact) clones from the reference set built in [3] belonging to the biggest project (eclipse-jdtcore) and intersected it with the duplication chains set found by DuDe. Concluding, our tool's recall was 89% under the strict conditions of the [3] experiment, but in a more loose context the recall could rise up to 95%.

3.3 Validation of Scalability

The largest project over which we successfully applied our approach, was a C project with 32 MB of source code and over 600,000 LOC. The analysis took 2h45m, which is an acceptable amount of time for such an industrial-size project.

4 Related Work

The idea of analyzing the non-matching parts of the duplication appeared in [10], where the authors used it to observe evolution between several versions of a system. An interesting contribution, similar to our approach was [11], whose authors address the gapped clones issue by combining the exact clones provided by their token-based clone detector [7]. Various other techniques for detecting clones have been proposed over the years: based on scatter-plots [1], [4] and [5], on metrics [9] or abstract syntax trees [2] and program dependency graphs [8].

[2]Recall is the percentage of discovered clones over existing clones

5 Conclusions

5.1 Pros and Cons

One of the major advantages of our approach is that it provides additional duplications to the ones detectable by other traditional methods. It is able to bring to light smaller duplication fragments, otherwise hardly noticeable, which belong to a bigger, thus more important duplication block. By doing these, it ensures that the refactoring decisions are made with improved comprehension of the big picture, *i.e.*, it provides support for proper refactoring. Furthermore, the flexibility provided by means of the thresholds can lead to customized detection methodologies, that fit particular maintenance focuses.

As for the drawbacks, the tool is not capable of detecting renamed variables, due to its rather high granularity of comparison, although some of those could be found under modified duplication chains (with lower precison).

5.2 Future Work

To address the problem of course granularity, we think that a fuzzy comparison would make an improvement, giving the tool the advantages of a classical token-based duplication detector along with its novel duplication chain recovery approach. We would also be interested in researching on the information we could extract from the signatures of code duplication chains. Finally, while we think that it would be possible to associate some patterns in the signatures of duplication chains, it would be a real challenge to provide some assistance in the refactoring process, based on some identified patterns.

6 Acknowledgments

This work is supported by the Austrian Ministry BMBWK under Project No. GZ 45.527/1-VI/B/7a/02. We would also like to thank the LOOSE Research Group (LRG) for being such a great and challenging team.

References

[1] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[2] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings ICSM 1998*, 1998.

[3] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, Universität Stuttgart, September 2002.

[4] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines for text and code. *J. Computational and Graphical Statistics*, 2(2):153–174, June 1993.

[5] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM '99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, September 1999.

[6] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[7] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[8] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eigth Working Conference on Reverse Engineering (WCRE'01)*, pages 301–309. IEEE Computer Society, October 2001.

[9] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software System Using Metrics*, pages 244–253, 1996.

[10] F.Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *International Workshop on Principles of Software Evolution (IWPSE)*, pages 126–130, 2003.

[11] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *Proceedings Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, pages 327–336, Gold Coast, Australia, December 2002. IEEE.