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 don't intend to emphasize the consequences of introducing duplicated code anymore.

In order to ensure that the code says everything once and only once, the duplicated code has to be refactored. Spotting it can sometimes be obvious, but most of the time it is more subtle or easy to miss, especially with industrial-size software systems. Detection and analysis of the code duplication in legacy systems without powerful and reliable tool support is hard to imagine. This is why we consider that detecting clones and quantifying them is essential for further design improvement.

Throughout the last decade, there have been many approaches to detect duplicates. Some of the actual tools are able to detect slightly adapted code (variables, constants and methods renaming), while still overlooking larger-scale adaptations, *i.e.*, inserted or deleted lines of code.

In a typical lightweight line-based approach, large blocks of code affected by various modifications (from renaming operations to statement insertions or removals) would wrongly 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 merges such small fragments that belong together, thus providing the maintainer with some additional duplication blocks, otherwise granted with less importance or not detected at all (due to filtering).

1.1 Outline

The paper is further structured as follows: Section 2 presents the concept of duplication chain and other related terms, placed in the archeology metaphor context. Section 3 is a detailed presentation of our approach. Section 4 walks the steps we took to validate the approach, proving that it brings real benefits to maintenance. Section 5 will make a brief description of the related work and current concerns in this field of research, and finally Section 6 will point out the advantages and disadvantages of the presented approach, ending with a short presentation of our future work.

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 every scattered duplicated block, in order to make the right refactoring decisions.

2.1 It started with a Scatter-Plot

The scatter-plot approach was successfully applied in the code duplication detection field starting with the early '90s [1], [4], [5]. These approaches provide the maintenance engineer with textual information (sets of longest matches), but mainly with a visual representation in form of a scatter-plot (or dotplot), which can draw attention over "grey" areas of the matrix, possibly hosting duplicated code. From here, the specialist would further investigate the results.

The scatter-plot inspired us in the first place and our algorithm is based on this visual concept. It also provides an easy way to present the ideas behind our approach throughout this paper.

2.2 Need for duplication chains

Imagine we have the two pieces of Java code from Figure 1, which is a trivial example of scattered duplication belonging to a single duplicated block. 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){
  t[i]=tSensor[i].getTemp();
  lcd.println("T"+i+"="+t[i]);
  i++;
}
regulateTemp(temp);
```

```
initSensors(tSensors);
readSensors(tSensors);

int i = 0;
while(i<tSensors.length){
  temp[i]=tSensor[i].getTemp();
  System.out.println("T"+i+"="+t[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 pairwisely by non-matching gaps. Figure 2 illustrates the previous example's scatter-plot representation, where each marked cell corresponds to a match between the pair of lines of code intersecting in that precise point.

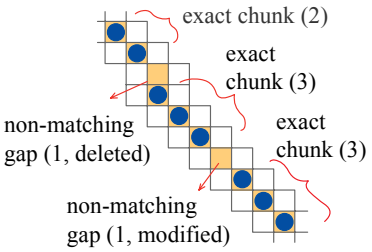

Figure 2. Duplication chain

An *exact chunk* is a non-altered part of a duplicated block, or in the context of the archeology metaphor an artifact found in its place. Exact chunks appear in a scatter-plot as continuous diagonals, as it can be seen in 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).

Two closely related characteristics of a duplication chain, directly influenced by the adaptation process are the *type* and the *signature*. The *type* of the duplication chain provides a summary of the adaptations made to the duplication block. We identified the following types of duplication chains: exact (regular clones, which did not suffer any adaptation), modified (duplication chain made of exact chunks linked by gaps composed of modified lines of code), insert/delete (chain with insert/delete gaps) and composed (exact chunks linked by various gap types).

The *signature* further extends the meaning of the type by capturing the structural configuration in terms of exact chunks, non-matching gaps and the metrics around them. Placed in the archeological context, 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, all of them measured in number of lines of code (LOC):

- *Size of Exact Chunk* (SEC) is a key metric that reflects the degree of the granularity left behind by the adaptation phase of the copy-paste-adaptation process. Furthermore, SEC is closely related to how painful the refactoring to eliminate this duplication could be.
- *Line Bias* (LB) is the size of a non-matching gap between two consecutive exact chunks. 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. The lower the distance (LB), the higher the probability that the two exact chunk are part of the same duplication block and possibly the higher the refactoring potential.
- *Size of Duplication Chain* (SDC) is the size of the more meaningful block of duplication, which actually suggests its magnitude. In the particular case of an exact type duplication chain, SDC is the same as SEC.

In order to constrain the duplication chain's proportions, we will impose the following thresholds:

- a minimum SDC, whose task is to ensure that the total length of the duplication chain is large enough to qualify it as significant,
- a minimum SEC, in order to avoid detecting duplication chains containing "duplication crumbs" *i.e.*, very small duplicated code fragments,
- a maximum LB, which quantifies the "neighborhood" aspect as it will make sure that the consecutive pieces of the chain are not too far from each other.

Since it is not desirable to detect duplication chains with gaps larger than its exact chunks, we should also make sure that: minimum SEC $\geq$ maximum LB.

There is no such things as a perfect threshold value. Still, from our experience we found that the following threshold values are generally adequate: minimum SDC = 8, maximum LB = 2 and minimum SEC = 3. The minimum SDC of 8 is justified because we considered that a significant duplication chains should be larger than the minimum configuration duplication chain of 2 exact chunks with SEC of 3, separated by a minimum length (LB of 1) non-matching gap.

3 Approach

In this paper, we propose a lightweight line-matching approach, enhanced with the concept of chain duplication, which can also cover duplications that cannot be detected by regular line-matching approaches.

3.1 Stepwise Recovery Methodology

Our approach walks the first steps of a usual scatter-plot approach, enhancing it with an additional step which builds the duplication chains. The phases of our detection process are:

Phase 1: Code Preprocessing The first phase starts by reading the source-files line by line, eliminating the white spaces, so that the various indentation styles would not make any difference. Then we eliminate noise (*i.e.*, lines of code made of syntactic elements like a single closing brace or empty lines), which is defined in a file in form of a set of regular expressions. An optional feature, and at the same time the only language-dependent part of our approach, is the possibility to ignore comments in the analysis process. This phase provides a set of *relevant* (noise free) lines of code in a raw form (without white spaces).

Phase 2: Populate the scatter-plot As in a regular scatter-plot approach, we compare every line of code (specifically, relevant code) with every line of code in the project. Every matrix cell *M[i,j]* will store the result of the comparison between the relevant line *i* and the relevant line *j*. As a result of this comparison, the matrix will be divided in two symmetric areas, around the main diagonal. Our approach works with only one half of the matrix (excluding the diagonal) to avoid storing redundant information.

Phase 3: Build the duplication chains Starting with the upper left corner of the matrix, we look for the first marked cell. This is a starting point for a potential valid exact chunk (and in the same time, for a duplication chain), which will be further extended up to the first unmarked cell on the main diagonal direction. The current exact chunk will be considered a valid one if its size exceeds the minimum SEC threshold, in which case the exact chunk is linked to the end of the current duplication chain candidate. Then we continue searching in the vicinity for another exact chunk, by increasing our "sight". If another valid exact chunk is found nearby while the maximum LB threshold was not exceeded, it is added to the duplication chain candidate. In the end, when the expansion of the chain candidate is not possible anymore, its size is measured and if it proves to be larger than the minimum SDC threshold, the candidate is declared

valid and it is added to the list of found duplication chains (results). In the same manner the search is repeated until we reach the lower right corner of the matrix.

A scatter plot representation, augmented with the compared lines is illustrated in a more complex example, an adapted duplication that summarizes all the types of operations on the lines of code (Figure 3).

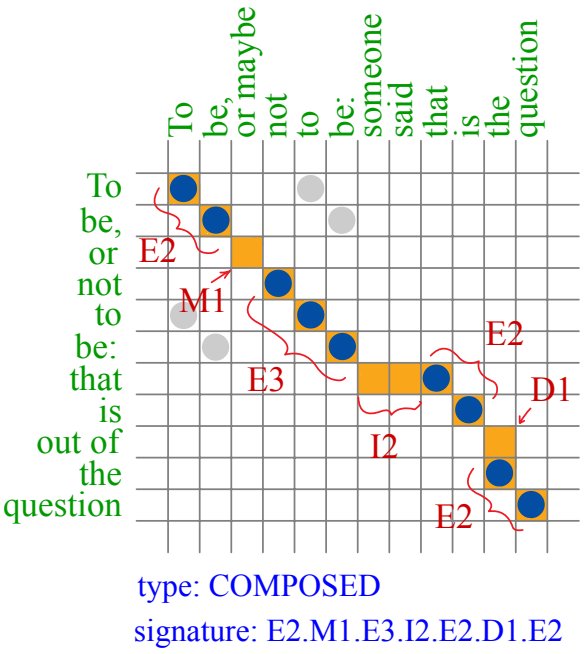

Figure 3. Complex duplication chain

3.2 Tracking the duplication chains

For an even better understanding of our methodology, we will present a pseudo-code description of the main algorithm used by the proposed approach for the detection of code duplication chains:

```
program DuplicationDetection
 for every entity E[i] in the project
  for every entity E[j], with j>i+1
     Area := area between E[i] and E[j];
     PopulateScatterPlot(Area);
     L := BuildDupChains(Area);
     add list L to ResultList;
  end for
 end for
 return ResultList;
end program
```

In order to increase the scalability, instead of processing the whole matrix at one time, we split the matrix into areas, based on pairwise intersecting entities (*i.e.*, source files or method bodies).

```
procedure PopulateScatterPlot(Area)
 for every line L in Area
  for every column C in Area,
              starting with L+1
    if code(L) matches code(C)
       then mark M[L,C] as unused;
    end if
  end for
 end for
end procedure
```

Every line of the matrix is compared to every one of its columns. Every cell representing a match between the lines of code that intersect in that specific point is marked as "unused match", as long as it has not been part of a duplication chain.

```
procedure BuildDupChains(Area)
 for every line L in the Area
  for every M[L,C] marked as unused
    create list of coordinates CList;
    add M(L,C) coordinates to CList;
    while GetNextCoordinate(C,L) is valid
        add its coordinates to CList;
        update SEC
    end while
    if SEC < minimum SEC threshold
        remove the coordinates of the
            last exact chunk from CList;
    end if
    if SDC(CList) > minimum SDC
        chain := CreateDupChain(CList);
        add chain to LocalResultList;
    end if
  end for
 end for
 return LocalResultList;
end procedure
```

The **GetNextCoordinate** method searches the coordinate of a potential expansion point. If the next natural cell is marked as "unused match" it returns its coordinate. Else, it searches further by stepwise increasing the LB until either a valid coordinate was found or the LB reached the maximum LB threshold. In the latter case, there is no next valid coordinate.

The **CreateDupChain** method creates a duplication chain out of a list of coordinates, tracing its signature and every other characteristic. Finally it marks the cells in the list as "used matches", to avoid reusing them for building other duplication chains.

4 Validating the Approach

In order to prove the efficiency of our approach, we applied it against several open-source projects. Our purpose was to demonstrate the advantages over traditional code duplication detectors. Besides presenting the tool itself, we will discuss the results of our experiment.

4.1 Tool Support

We designed and implemented an automated line-based language-independent code duplication detector, called DuDe (Figure 4).

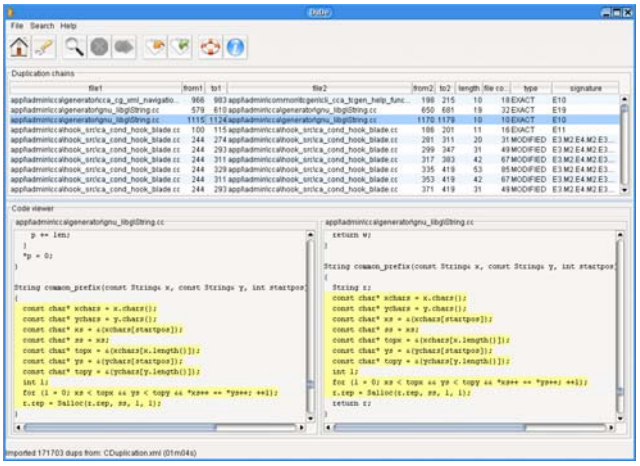

Figure 4. DuDe (Duplication Detector)

This non-visual tool, based on a visual model we call enhanced scatter-plot can be used in two contexts: either as a stand-alone tool, in which case its input consists of source files or as part of an integrated platform for quality assessment of object-oriented design called iPlasma [9], in which case a set of method bodies is provided as input.

Its flexibility is conferred by the tunable parameters, which can widen or narrow down the results of the detection, depending on the purpose of the analysis. To filter the results of the detection, the *minimum SDC* parameter should be increased, until the "noise" (short duplicates, less interesting for the refactoring phase) is bearable. The upper limit for the distance between two consecutive exact chunks within the same duplication chain can be modified by tuning the *maximum LB* parameter, while the size of the exact chunks can be imposed by way of *minimum SEC* parameter.

Although the examples throughout the paper are presented in a scatter-plot representation, instead of presenting the problem areas in a graphical representation like a regular scatter-plot approach, the tool will provide a list of duplication chains. This allows the engineer to jump to the analysis of these candidates by means of the duplicated code visualization feature, rather than picking them first. Moreover, the list can be sorted using multiple criteria (entity, starting line, ending line, type, signature, size), thus providing the means for further interesting analyses.

The duplicated code for the selected chain is presented in highlight in the code visualization panel, an important feature for the process of manual validation of the results (which we extensively used in our experiments). Furthermore, the results can be exported in XML format and then imported, a useful feature for analysis session resuming.

4.2 Experimental Setup

For the first experiment, we chose 4 Java and 4 C projects, the same study cases from Bellon's paper on evaluation of clone detecting tools [3].

The 8 projects, covering the size range from 0.5 MB to 10 MB, are presented in Figure 5, along with the experiment's results. The table is composed of the projects presentation and the comparison between the results obtained by our approach and the ones that a regular line-based approach would have provided. For every project we included its name, the programming language used (PL), number of files (NOF), size on disk and number of thousands of lines of code (KLOC).

The purpose of the experiments was to answer questions regarding four precise issues:

1. Quantitative issue: is there any gain of duplication provided by our approach over regular line-based clone detectors?
2. Qualitative issue: how relevant, in the refactoring context, are the additional duplications?
3. Reliability issue: how reliable are the results, in terms of precision and recall?
4. Scalability issue: does our approach scale up?

4.3 Quantitative Validation

We established the minimum SDC of 7 LOC for both configurations. Then, we tuned DuDe to simulate a regular line-based code duplication detector, unable to recover duplication chains, by disabling the duplication chain building feature. To do this, we just set a maximum LB of 0 (no linking between exact chunks).

In the second configuration, we kept the minimum SDC of 7, which we further combined with a maximum LB of 2 and a minimum SEC of 2, enabling the recovery of duplication chains.

The results obtained with the two configurations are presented in terms of number of duplication chains (NODC)

					Regular approach		Our approach	
Project Name	**PL**	**NOF**	**Size (MB)**	**KLOC**	**NODC**	**COV (%)**	**NODC**	**COV(%)**
weltab	C	65	0.43	11	759	72	711	76
cook	C	590	2.68	80	1,285	9	1,744	16
snns	C	420	4.82	115	47,930	16	53,274	21
postgresql	C	612	9.52	235	704	8	1,070	11
netbeans-javadoc	Java	101	0.68	14	39	12	48	15
eclipse-ant	Java	178	1.43	35	14	2	24	4
eclipse-jdtcore	Java	741	6.90	148	716	12	1,127	16
j2sdj1.4.0-javax-swing	Java	538	8.39	204	1,171	7	1,388	10

Figure 5. Experiment Results

and *coverage*[1] (COV). After comparing the result tables we observed that:

- judging by NODC, it is obvious that our enhanced approach provides in almost every case more duplications
- COV is always larger in the case of our approach. A proof that this approach provides additional duplications (chains) is the increase of coverage from the traditional to the duplication chain approach. The duplication chain detection has higher coverage because it recovers larger duplications out of smaller chunks of duplication, overlooked by the first approach. In some extreme cases, the coverage detected by the second approach is twice as high as the coverage provided by the first one.

Concluding this experiment, we are able to answer the first question by stating that our "archeological" approach does offer an important amount of otherwise lost code duplication information.

4.4 Qualitative Validation

For our second experiment we picked an open-source software system called JHotDraw. We ran the tool against the JHotDraw source code and simulated a search with a tool without support for duplication-chains (minSDC = 8, maxLB = 0) and we obtained 42 duplication chains, all of type exact. Then we ran the tool with minSDC = 8, maxLB = 2, minSDC = 3. We obtained 72 duplication chains of different types. It is clear that we got 30 extra duplication chains in our approach, a fact that confirms once more the quantitative gain. Then, we extracted the 30 new found duplication chains and analyzed them one by one.

Summing up this experiment, we found that in 76 % of the cases the found duplication were relevant, having a high refactoring potential (*e.g.*, sets of methods belonging to related classes with a common superclass). These results reassures us that it makes sense to recover duplication chains, since the clone detection is often just an intermediate step towards refactoring.

4.5 Reliability Validation

The clone detecting tools are evaluated in terms of *precision*[2] and *recall*[3]. Perfect recall means all duplications are found, while perfect precision means no false positives are reported. In order to calculate the precision and recall of our approach, we need a reference set of clones for a certain system, which is a high desiderate and really hard to achieve, as stated in [15]. Such a reference set could be built by running a perfect tool on the system and storing the results. But we all know such a tool does not exist.

The solution adopted by Bellon [3] in his experiment was to merge the results provided by the clone detectors he compared, and to leave to a group of human arbiters the task to "manually" filter these results. Besides the fact that only 2 % of the data was reported at the end of that experiment, taking that set as a reference could lead us to some problems: if our tool is able to find clones not found by any of the tools that participated at building the reference set, these new and valid clones would not be present in the reference set. The precision is strictly related to this aspect.

What we found fair was to take out the reference duplications belonging to the biggest project (eclipse-jdtcore) and to run our tool on that project. After that, we would search the clones in the reference set in our results. We chose type 1 clones (exact clones) to test the recall of our tool on that

[1] Coverage is the percentage of lines of code in the system affected by duplication

[2] Precision refers the percentage of correct results

[3] Recall refers the percentage of the clones that are found

type of clones. Out of 120 clones, our tool found 107 of the clones, 7 of the clones were covered by larger clones found by our tool, and 6 were missed. After analyzing the missed clones, it was clear that these clones were detected by token-based detectors (finer granularity) and cannot be detected by any line-based detector.

Concluding, under the strict conditions of Bellon's experiment [3] our approach had a 89 % recall, but in a more loose context (also considering the 7 clones found in larger duplication chains) the recall could rise up to 95 %.

4.6 Scalability Validation

For this purpose we present you some data on the scalability of the tool and on time performance. The largest project we successfully run our tool against was a project of 74 MB of C source code containing over 800,000 LOC. The experiment has been conducted on a machine with an Intel Pentium 4 processor at 2.8 GHz, with 1 GB of RAM. The detection process lasted about 4 hours, which is an acceptable amount of time for such an industrial-size project. This project was not included in the test-case validation due to non-disclosure reasons.

5 Related work

Software duplication has been a focus of research for at least a decade and dozens of papers on the topic have appeared.

Baker, in her 92 paper [1] proposed a tool called Dup with a line-based approach, which offers visual information (scatter plot) and can also look for parameterized matches (variables, constants).

Church and Helfman [4] proposed Dotplot, a visual tool that works with tokens of various granularity (words, lines). The observations made on different visual patterns are basis for most of the duplication chains types that we identified.

Mayrand *et al.* [10] proposed an interesting approach for detecting code duplication: based on metrics extracted from the source code by another tool (Datrix), they established function similarities.

A novel approach based on abstract syntax trees was proposed by Baxter *et al.* in [2], which can produce macros bodies to eliminate duplication. Due to its internal representation (ASTs), this tool is strongly language dependent, can be run only against compilable systems and has higher memory requirements than our lightweight approach. It is also able to find what we call modified duplication chains, but it misses our insert/delete or combined duplication chains.

Duploc [5] is a line-based approach visual tool that provides the scatter plot as output and can perform pattern matching over it to detect clones. It can also detect some clones similar to our modified duplication chains (but with only one gap in the middle). However, it misses our insert/delete and combined duplication chains. Moreover, there were some scalability problems reported in [3] with larger systems.

Another interesting approach was Krinke's [8] one, which is focused on detecting maximal similar subgraphs in fine-grained PDGs (program dependence graphs). Still, as he states, this approach cannot analyze big programs due to the limitations of the underlying PDG-generating infrastructure.

An interesting contribution was the one that Kamiya *et al.* proposed in [7]. Their token-based tool, called CCFinder is based on suffix tree matching algorithms. While the tool itself is not able to find gapped clones, in [14] the authors address this issue, supported by their maintenance environment called Gemini [13]. By combining the exact clones provided by CCFinder, Gemini proves to be more precise than any line-based approach (due to its finer level of granularity) with the price of language dependency.

The idea of analyzing the non-matching parts of the duplication also appeared in [12], where the authors used it to observe evolution between several versions of a software system.

Regarding the concern towards validation in terms of recall and precision of clone detecting tools, Bellon [3] conducted an experiment, whose main concern was to compare the quality of the results provided by several tools ([2], [7], [8], [5] and [10]). As a conclusion to that experiment, the author stated that there was no absolute winner, every approach implying both advantages and disadvantages.

6 Conclusions and Future Work

6.1 Pros and Cons

One of the major advantages of the approach presented in this paper is that it provides additional duplications to the ones detectable by other traditional methods. Moreover, it brings to light smaller duplication fragments, otherwise hardly noticeable, which belong to a bigger, more important duplication block. By doing this, it ensures that the refactoring decisions are made with improved comprehension of the big picture, *i.e.*, it provides support for proper refactoring. The support tool proved some industrial strength it terms of scalability and language-independence. Furthermore, the flexibility provided by means of its parameters 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 these are covered by duplication chains of type modified.

6.2 Future Plans

With the disadvantages of this approach in mind, we will formulate some possible directions of improving this solution. First, to address the problem of coarse granularity, we think that a fuzzy comparison would make an improvement. Instead of a boolean result, the comparison would provide a fractionary result, which would answer the question "How much are they alike?". With such a comparison method, the tool would combine the advantages of a classical token-based duplication detector with the novel duplication chain recovery approach.

Introducing complementary visualization representations of the duplication chains would be another priority on our list. In this context, there is an ongoing interest on advanced visualization techniques for code duplication [11].

In extension to these desiderates, we would be interested in researching on the information we could extract from the signatures of code duplication chains and to provide assistance in the refactoring process, based on identified patterns.

7 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] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Raţiu, and R. Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, September 2005.

[10] 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.

[11] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109, 2004.

[12] 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.

[13] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Eighth IEEE International Symposium on Software Metrics (METRICS'02)*, pages 67–76, Gold Coast, Australia, June 2002. IEEE.

[14] 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.

[15] Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia. Problems creating task-relevant clone detection reference data. In *WCRE*, pages 285–295, 2003.