Lightweight Detection of Duplicated Code — A Language-Independent Approach[1]

Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger[2]

IAM-04-002

February 12, 2004

Abstract

Duplicated code can have a severe, negative impact on the maintainability of large software systems. Techniques for detecting duplicated code exist but they rely mostly on parsers, technology that is often fragile in the face of different languages and dialects. In this paper we show that a lightweight approach based on simple string-matching can be effectively used to detect a significant amount of code duplication. The approach scales well, and can be easily adapted to different languages and contexts. We validate our approach by applying it to a number of industrial and open source case studies, involving five different implementation languages and ranging from 256KB to 13MB of source code. Finally, we compare our approach to a more sophisticated one employing parameterized matching, and demonstrate that little if anything is gained by adopting a more heavyweight approach.

1 Introduction

Duplicated code arises naturally during the development and evolution of large software systems for a variety of reasons. Duplication can have a severely negative impact on the maintenance of such systems due to code bloat, added complexity, missing abstraction, and the need to maintain multiple copies of nearly identical code [1]. Although duplicated code is conceptually simple, it can be surprisingly hard to detect in large systems without the help of automated tools.

Various approaches have been applied in practice with promising results [2, 3, 4, 5, 6, 7, 8]. The main technical difficulty is that duplication is often masked by slight differences: reformatting, modified code, changed variable names, and inserted or deleted lines of code all make it harder to recognize software clones. The general approach to combat this effect is to parse the code and compare the parsed structures. Although this technique avoids certain problems, it is heavyweight, and, more importantly, it is brittle, since the approach must be adapted to every programming language and dialect under consideration [7]. This is clearly stated in the following quotation:

> *"Parsing the program suite of interest requires a parser for the language* dialect *of interest. While this is nominally an easy task, in practice one must acquire a tested grammar for the dialect of the language at hand. Often for legacy codes, the dialect*

> *is unique and the developing organization will need to build their own parser. Worse, legacy systems often have a number of languages and a parser is needed for each. Standard tools such as Lex and Yacc are rather a disappointment for this purpose, as they deal poorly with lexical hiccups and language ambiguities."* [7].

In summary, most of the approaches [3, 4, 6, 7, 8] are based on parsing techniques and thus rely on having the *right* parser for the right dialect for *every* language that is used within an organization.

Instead, we propose a lightweight approach based on simple string-matching. We cope with differences in formatting by means of a pre-filtering stage that can easily be adapted to different languages and conventions. Comparison by string-matching is straightforward, and is readily implemented by standard libraries and tools. Sensitivity to changes in the duplicated code segments is easily addressed by considering minimum sequence lengths and density of duplicated code.

We have applied the technique to a large number of case studies. The approach works well at detecting most of the duplicated code, and also scales well. Experiments show that certain standard sensitivity settings are best for most of the case studies.

Further experiments based on parameterized string-matching [9, 10], which approximates the more sophisticated parser-based approaches, show that attempts to refine the technique only identify at most 8% more duplication. Our study shows that the *upper bound* limit of not identified duplicated code using simple string matching is never more than 8%, and the average is under 5% for duplicated code sequences at least 10 lines in length.

In section 2 we provide a brief overview of the approach. In section 3 we validate the approach in terms of adaptability, tunability, scalability, and coverage. In section 4 we compare the quality of the lightweight approach against that of parameterized string matching. In section 5 we discuss related work. In section 6 we conclude with some remarks on future and ongoing work.

2 Detecting Duplicated Code by String-matching

Although the notion of duplicated code is intuitively clear, the problem of detecting it is not so well-defined. Consider the following requirements for

detecting duplicated code:

Multiple source files. Code may be duplicated within a single file, or across multiple source files.

Avoid false positives. False positives occur when code is marked as duplicated that should not be. Programming language constructs, idioms, and recurring statements should not normally be considered as duplicated code, since they do not indicate copy-and-paste problems.

Avoid false negatives. False negatives arise when code is duplicated, but slightly altered in such a way that it is no longer recognized as being a clone. A good code duplication detection tool will be robust in the face of insignificant or minor alterations in the duplicated code. The real difficulty is to be precise about when two similar pieces of code should be considered duplicates of one another or not.

Scalability. Duplicated code is most problematic in large, complex software systems. For this reason, a useful tool must be able to cope with very large code bases.

Multiple languages and dialects. There are thousands of programming languages in use today, and hundreds of dialects of the most popular languages (like C++). A useful duplicated code detector must be robust in the face of syntactic variations in programming languages, and should be configurable with a minimum of effort. In particular, a tool that can only be configured by an expert in parser technology is unlikely to be popular.

The approach we advocate basically consists of the following three steps:

1. *Eliminate noise:* transform source files into *effective files* by removing comments, white space, and other uninteresting artifacts,

2. *Compare the transformed code:* compare effective files line-by-line, and

3. *Filter and interpret the results:* apply some simple filters to extract just the interesting patterns of duplicated code.

Multiple sources files are easily compared by this technique. False positives are avoided by removing certain kinds of noise and by filtering. False

negatives are similarly avoided by removing noise that will perturb the comparisons. The approach scales well due to its simplicity. Finally, the approach is easily adapted to different languages since it does not rely on parsing. Noise reduction may have to be adapted for each language, but this is much simpler than adapting a parser.

The results can be reported in a variety of ways. Duploc is an experimental platform that visualizes the comparison matrix as a dotplot [11]. These visual displays are linked to the source code to support reverse engineering. The results can also be used as input to a refactoring engine that can eliminate duplicated code by introducing the missing abstractions [12].

In the rest of this section we will describe the technique in some more detail. In the following sections we will validate our claims in the context of a significant number of case studies.

2.1 Noise Elimination

Noise elimination serves two purposes. It reduces false positives by eliminating common constructs and idioms that should not be considered duplicated code. It also reduces false negatives by eliminating insignificant differences between software clones.

What should be considered noise depends not only on the programming language, but also on what information you want to extract. Typical operations include:

- the elimination of all white space and tabulation,
- the elimination of comments, and
- the elimination of uninteresting language constructs.

Other operations that could be performed, depending on the programming language in question, include:

- the removal of all preprocessor directives,
- the removal of all block and statement delimiters, and
- the conversion of all characters to lower case.

For example, the following code snippet:

```
#include <stdio.h>

static int stat=0;

int main(void)
{
int local=1;
int *dynamic = (int*) malloc(sizeof(int),1);
```

might be transformed to:

```
staticintstat=0
intlocal=1
int*dynamic=(int*)malloc(sizeof(int),1)
```

Noise elimination is easily specified as a short program in a dedicated text manipulation language, such as `perl` [13].

2.2 Comparison

After noise is eliminated, effective files are compared to each other line-by-line. The naive comparison algorithm is $O(n^2)$, but this is easily improved by hashing lines into B buckets, and then comparing only lines in the same bucket [7].

Lines are compared for exact string matches. The result of the comparison is a matrix of hits and misses. Figure 1 illustrates a typical matrix visualized as a dotplot [14, 15]. Line numbers increase downwards for one file, and to the right for the second file. In this example, we see that the second file is not only longer than the first, but more than half of the file essentially duplicates the first file. This is manifested by the long diagonal of hits.

Exact clones are relatively rare. Instead, code is more typically duplicated and then modified in various ways. Figure 2 illustrates some typical duplication scenarios:

a. pure duplication results in unbroken diagonals.

b. modified lines appear as holes in the diagonals.

c. deletions and inserts result in broken, displaced diagonals.

d. case statements and other repetitive structures yield grid-like, rectangular blocks.

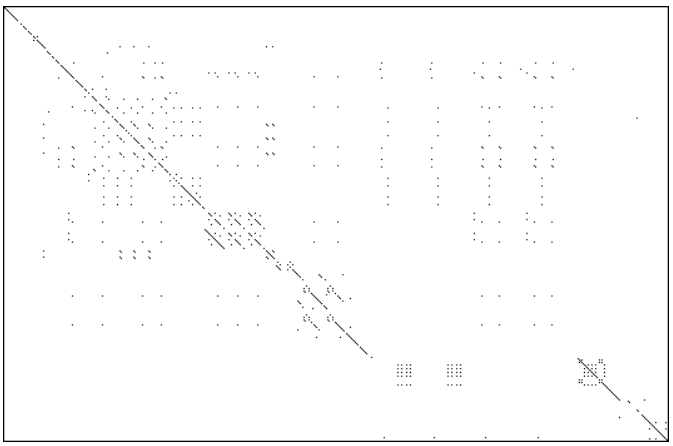

Figure 1: A dotplot of a comparison matrix applied to a file and a more recent version of itself.

The comparison matrix will typically contain many hits, some of which are significant, and others which are not. In order to extract the significant hits, we perform a third, filtering pass.

2.3 Filtering

We are interested in detecting significant code duplication, but how do we determine what is "significant"? We can interpret this in several ways.

First of all, we are not so much interested in individual lines of code that may be duplicated, but rather in longer sequences that may contain a certain amount of duplication. We call these *comparison sequences*. We quantify the duplication in two comparison sequences by considering either the *gap size*, *i.e.*, the length of non-duplicated subsequences, or the *duplication density*, *i.e.*, the ratio of duplication hits to the total length of the comparison sequence.

For example, if we compare the sequences "abcdefghi" and "abcdxyzefg", we find a gap of length 3, and an overall duplication density of 66% (6 hits in sequences of length 9).

This leads us to consider the following filter criteria:

1. *Minimum length*. This is the minimal length for a comparison sequence to be considered "interesting".

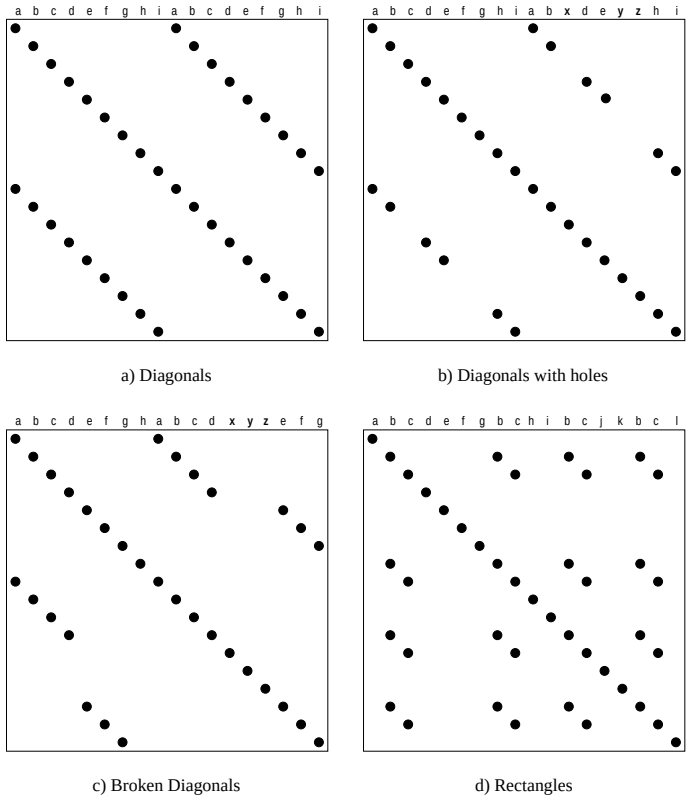

Figure 2: Dotplot visualization of typical duplication scenarios.

2. *Maximum gap size.* This is maximum allowable gap size for sequences to be considered duplicates of one another.

3. *Minimum density.* This is the minimal duplication density for comparison sequences to be considered duplicates of one another.

Filtering is purely concerned with eliminating false positives, since filters only remove duplicates detected in earlier phases.

In our experiments, a minimum length of 10 lines proved to be good for detecting significant duplication. Low values, like 1 or 2, generate too much noise, and large values, like 30, filter out virtually all code duplication.

A maximum gap size of 4 picks up most cases of "interesting" duplication. A small gap size of 0 will only pick up exact clones, which is not so interesting.

Filtering based on minimum density is used similar effect, but is less sensitive to the length of the duplication sequence. Note that a maximum gap size of 0 forces the minimum density to be 1, and vice versa. In practice, a minimum density of around 50% will filter out uninteresting cases of duplication.

We discuss the impact of the filter settings in more detail in sections 3.3 and 3.4.

3 Case Studies

In this section we explore how well the string-matching approach works in practice by considering a number of case studies. The case studies have been chosen to cover a variety of different application characteristics:

- Different programming languages, including C, C++, Java, Smalltalk, Python and Cobol.
- Different development environments, including industrial code, open-source, and academic research prototypes.
- Different sizes of projects, ranging from small, individual programs (*e.g.*, patch, 80KB C code) to large applications (*e.g.*, GNU CC, 13.4 MB C), and small to large numbers of program files (*e.g.*, Database Server, with 593 classes in individual files).

An overview of the case studies is given in the appendix A.

3.1 Language Independence

The string-matching approach is easily adapted to different programming languages in a matter of minutes. The only aspect which should normally be adjusted is the noise elimination, since white space and comments are different for each programming language in question, and may affect both the number of false positives (*e.g.*, duplicated comments) and false negatives (*e.g.*, duplicated code with modified indentation).

Noise elimination is typically achieved by means of a simple script in a dedicated text-manipulation language, such as `perl`, `awk` or `sed`. All that is needed is a simple scanner which is able to recognize delimited text, *i.e.*, multiline comments. We have implemented a fully generic and fairly complete

solution which also handles strings in about 150 lines[3] of `perl` code. A simple parameterization of the relevant delimiter symbols then suffices to ready the tool for any new language. Small extensions may be necessary for programming languages which require a strict source line format, *e.g.*, COBOL.

3.2 Scalability

String-matching is cheap and efficient even for very large amounts of source code and large amounts of duplication. The largest case study that Duploc was applied to was GNU CC (13.4 MB of source code), and the largest degree of duplication detected was 59% (in a payroll system).

In practice, the string-matching itself is not a bottleneck, since the complete source code must be processed only once, and this may be done in a single batch job. (Noise elimination may need to be fine-tuned, but this is normally done with a small subset of the source code.) The real difficulty is to interpret the large amounts of data that are generated.

Dotplots work well to get an overview of a few thousands of lines of code at a time, but are not adequate for exploring large systems. For large systems we developed on the one hand a mural-based visualization of dotplots and on the other hand various code duplication reports, but this is out of the scope of this paper as we focus here on the relevance of our approach compared with other more heavyweight techniques [11].

3.3 The Impact of Minimum Sequence Lengths

We measure duplicated code either within a single file or across a set of files in terms of the fraction of lines of code that occur more than once. If, for example, a sequence of 10 lines of code occurs twice in a 100-line file, then this file exhibits 20% code duplication (20 lines out of 100). Typical industrial code is estimated to contain about 8-12% duplication [5]. Duplication rates of 25% are considered exceptionally high, and rates of 50% or more are rarely seen in practice.

The actual degree of duplication will depend on noise elimination and filtering, since duplication without modification is rare. For the majority of the case studies, we experimented with the following settings:

[3]See Appendix B for a dedicated solution for C/C++ source code in about a dozen lines.

1. *Minimal Length:* We selected sequence lengths 10, 20, and 30 as minimum thresholds. We did not explore higher thresholds, since duplication fragments of length > 30 are highly unusual. From the point-of-view of reengineering, however, a threshold of around 20 could be useful as a rule-of-thumb for identifying opportunities for refactoring.

2. *Maximum Gap Size:* For all the case studies, we have set the gap size to 4. This is a high value, allowing up to four changed lines between two lines that must match exactly.

3. *Minimum Density:* We did not impose any density constraint, so the minimum density threshold was set to 0.

Clearly, increasing the minimum sequence length will decrease the proportion of duplicated code that is detected. Below, in section 3.4 we consider the impact of other settings for gap size and density.

3.3.1 Industrial Case Studies

The industrial case studies case studies we explored exhibited from 6% to 59% duplication. Although this may seem unusual, these case studies were provided to us precisely because they were suspected of suffering from high rates of duplication, and they should therefore not be considered truly representative.

The average duplication for this category, including all case studies, are:

Coordinate	*Average*	*Variance*
Sequence Length 10:	35%	0.0558
Sequence Length 20:	26%	0.0529
Sequence Length 30:	20%	0.0469

Note that if a fifth of the source code is duplicated — when only looking at long copied sequences of 30 lines or more — it must be considered a very high rate.

It is also remarkable that increasing the sequence length by a factor of three does not even halve the duplication rate (see the other categories for a comparison), clearly a sign of a very large proportion of copied code.

3.3.2 Open Source Case Studies

The open source case studies exhibited dramatically lower duplication rates than the industrial ones. The average duplication for the open source cate-

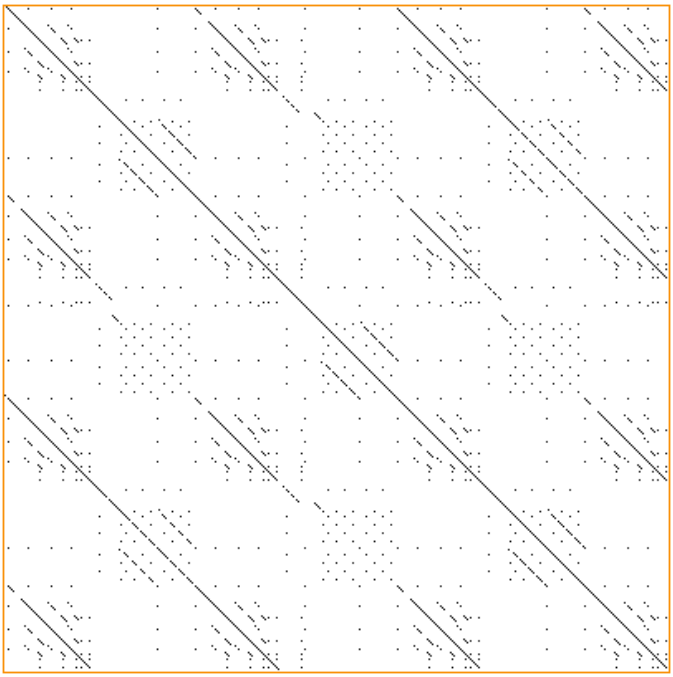

Figure 3: An extract from a comparison matrix of the `agrep` case study. We see that the code consists in regular repetitions of the same few lines, with only marginal variations.

gory is as follows:

Coordinate	*Average*	*Variance*
Sequence Length 10:	10.2%	0.0041
Sequence Length 20:	4.7%	0.0017
Sequence Length 30:	2.9%	0.0012

Note that the increase of the length constraint by a factor of two effectively halves the duplication rate. Increasing the sequence length again by a third reduces the duplication rate by 40%.

One of the open source case studies with the most duplication is the `agrep` [16] tool. Almost 25% of source lines are copied in sequences of at least length 10, and for length 30 we still have 14% of the code duplicated. When looking at the copied sequences, we see that the algorithm is expanded in multiple instances of the same steps. (see Figure 3).

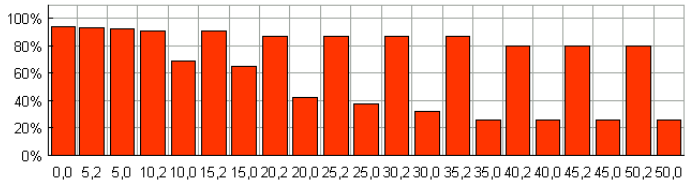

Figure 4: An example for the Impact of setting MaxGapSize to Zero. First number of X-Axis labels is MinimumSequenceLength, second number is MaxGapSize.

3.3.3 Academic Software Case Studies

The software produced in an academic context exhibits small to moderate duplication rates with only one case study going over 15% with any of the filter settings. There are a number of case studies for which we could only find around 5% of duplication at sequence length 10. In these cases, the duplication drops to under 2% for the bigger sequence lengths.

The numbers for this category are:

Coordinate	*Average*	*Variance*
Sequence Length 10:	12%	0.0063
Sequence Length 20:	5%	0.0034
Sequence Length 30:	4%	0.0019

We see that the largest part of the recorded duplication is for sequences of at least 10 lines, and since there is no big difference between lengths 20 and 30, the smaller part of the duplication is made up of long sequences.

3.4 The Impact of Gap Size and Density Settings

Using a gap size that is too small eliminates a great deal of duplication that would otherwise be detected. In Figure 4 we see the impact of setting the gap size to 2 and to 0 in the UVG case study, with a variety of different minimum sequence lengths. With a gap size of 2, duplication is detected at rates ranging between 80% and over 90%, even when the sequence length ranges from 0 to 50. With a gap size of 0, however, the duplication detected drops rapidly to just over 20% as the sequence length increases.

The density threshold removes comparison sequences that do not contain enough matches, but are nevertheless stretched by frequent mismatches or

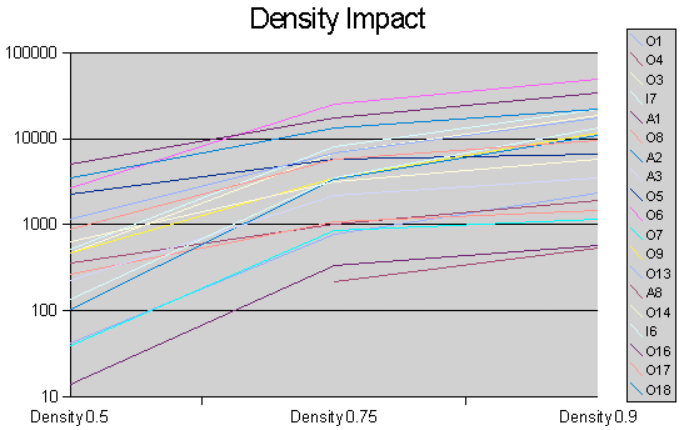

Figure 5: Lines of code suppressed by a specific density setting, compared to unrestricted density. Density 75% has the biggest impact in almost all case studies, when compared to the next lower setting (Values of Table 1). Note that the scale is logarithmic to account for the different sizes of the case studies

gaps. This happens when we have a repetitive piece of code, like for example a switch statement in C or C++, where we have the recurring line break;. If the contents of the individual branches of a switch statement consist of a single line, a comparison will produce a sequence where matches (the break; lines) and gaps (the code of the switch-branches) alternate. Depending on the length of the switch statement, the sequence can encompass quite a number of lines. To weed out such undesirable sequences, we use the density threshold.

In Table 1 we see that the amount of duplication in LOC that is filtered out increases as the minimum density increases (hardly a surprising result). These results are plotted in Figure 5. Note that the minimum density of 75% has the largest increase compared with densities of 50% and 90%.

4 Parameterized String-matching

Although simple string-matching clearly detects a great deal of duplication, it is natural to ask how much duplication it *misses*. The difficulty with this question is that "duplication" is in the eye of the beholder. Consider, for

Case Study	Density		
	50%	**75%**	**90%**
Agrep	43	773	2344
Bison	366	1014	1904
Apache	477	6756	18658
BlobSections	521	8206	22486
CodeCrawler	14	339	581
DiffUtils	269	1073	1482
Dome	105	3435	11080
Duploc	225	2161	3514
Flex	2264	5716	6649
GCC	2708	25579	49279
GnuJSP	40	862	1147
JBoss	464	3502	11708
SDL	1150	6786	17307
Tgen	0	218	535
Tomcat	628	3187	5907
UVG	138	3498	13382
XEmacsC	5149	17593	33813
XEmacsLisp	876	5821	9473
ZooLib	3510	13487	22422

Table 1: Amount of duplication (in lines of code) that is removed when using different density filter setting. The MaxGapSize Filter Setting was 2 for all of the data.

example, the two program fragments below.

```
int i;       int j;
i = 5;       j = 5;
i++;         j++;
```

The duplication is readily apparent to the eye, but would not be detected by simple string-matching.

To detect this form of duplication, it is necessary to extract the *structure* inherent in a program, and compare structures rather than literal strings. Parsing code and comparing the resulting abstract syntax trees, however, is an approach that is considerably more heavyweight than we wish to consider, for reasons we have already outlined.

Instead, we propose to estimate the theoretical *upper bound* of the duplication that we fail to detect by simple string-matching, by exploring the duplication detected by means of *parameterized string-matching* [2, 17]. The idea is simply to add a pre-processing phase which *abstracts from variable language constructs*, and then to perform simple string-matching on the transformed programs. In general this will produce many false positives, but it will also give us an upper bound to the duplication that would be detected by any more sophisticated approach based on comparison of ASTs or other structured representations of programs.

A simple way to achieve this abstraction is to use regular expressions to match elements of the source code that belong to a common syntactic category, and replace them by a string representing that category. For example, all identifiers may be replaced by a generic name like `p`. See the following Table for a list of other code elements that can be abstracted.

Language Element	*Example*	*Replacement*
Identifier	`counter`	`p`
Literal String	`"Abort"`	`"..."`
Literal character	`'y'`	`'.'`
Literal Integer	`42`	`1`
Literal Decimal	`0.314159;`	`1.0`
Function Name	`main()`	`foo()`

Note that the keywords of the programming language are not abstracted. This stems from the consideration that language keywords, which give the code its basic structure, must be the same in two code fragments if they are to be considered duplicated.

By choosing which elements in the above table to abstract and which not, we introduce an "abstraction level" into which the code is transformed. A particularly useful level is that which abstracts every element in the table above except function and method names. See Figures 6 and 7 for an example of this kind of transformation.

```
1   def manage_first(self, selected=[], REQUEST=None):
2     options=self.data()
3     if not selected:
4       message="No views were selected to be made first."
5     elif len(selected)==len(options):
6       message="Making all views first has no effect."
7     else:
8       options=self.data()
```

Figure 6: Python-source code from the Zope Application Server.

```
1   def manage_first(P, P=[], P=P):
2     P=P.data()
3     if not P:
4       P="..."
5     elif len(P)==len(P):
6       P="..."
7     else:
8       P=P.data()
```

Figure 7: The same source code as in Figure 6, after having abstracted all identifiers (except method names). Note that also literal strings have been abstracted.

The required transformation is implemented with moderate effort. It suffices to build a small tokenizer/lexer with a lookahead of one using regular expressions (see Appendix C). In this way, we determine for each token to which class it belongs and to which abstract replacement we have to change it. This approach is generic and was easily adapted for different languages (we have done so for C, C++, JAVA, COBOL and PYTHON). From this perspective, we claim that parameterized string-matching realized by the combination of abstraction and simple string-matching is still largely language-independent.

Parameterized string-matching not only detects more duplication than simple string-matching, it also produces more false positives. Consider, for example, the following code snippet and its abstraction:

```
n = 0;                          p = 0;
translations = _translations;   p = p;
CPnlast = onerecord[1];         p = p[1];
CPnfirst = onerecord[2];        p = p[1];
```

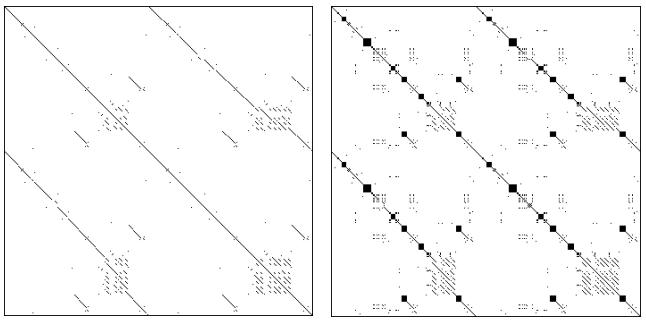

Figure 8: On the left, the matches of the unaltered source code. On the right, the matches that are produced by comparing the abstracted source code.

```
CPemail = onerecord[3];          p = p[1];
timestamp = onerecord[4];        p = p[1];
```

This clearly illustrates that all assignment statements of the same type will be flagged as duplicated code, which may result in large numbers of false positives. Consider, for example, Figure 8 which shows the duplication dotplots for both simple and parameterized string-matching of a program file. The main difference is the appearance of the black blocks at certain points in the program. These presumably arise due to sequences of assignment statements in the program. In this particular case, the dotplots quickly show us that no new duplication has been detected, but only false positives have been added. The black boxes only add noise.

In Figure 9, we see the relative difference between duplication found by parameterized string-matching and simple string-matching in our SMALLTALK case studies for sequence lengths 10, 20 and 30. We see that the amount of additional duplication detected by parameterized string-matching (possibly including false positives) is never more than 8%, and the average for sequence length 10 is under 5%.

The most striking detail of the chart in Figure 9 is that there is a notable difference in detected duplication only at the smallest sequence length of 10. For both longer sequence lengths, 20 and 30, the difference is almost always below 1%, the average difference being 0.8% for sequence length 20 and 0.3% for sequence length 30.

Note that we did not check for false positives among the additional duplication that was reported for the abstracted code, so the numbers reported

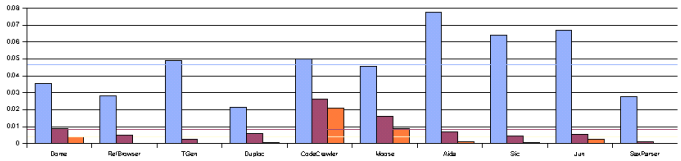

Figure 9: Differences between the amount of duplication found by parameterized string-matching and simple string-matching, at sequence lengths 10, 20, and 30 for each case study (SMALLTALK case studies).

here should only be considered as an *upper bound* for the duplication undetected by simple string-matching. This upper bound, however, is quite small in practice, so we conclude that simple-string matching is highly cost effective, and can generally be preferred to more sophisticated techniques.

5 Related Work

Many different techniques have been applied to identify copy and paste plagiarism [18, 19, 20, 21]. Techniques used include: structural comparison using pattern matching [4], metrics [5, 6], statistical analysis of the code, code fingerprints [22, 23, 3], or AST matching [7], slicing of program dependence graphs (PDGs) [24],[25], or a combination of techniques [26]. However, all these techniques share a generic process comprising of two steps:

1. Source code transformation. The source code is transformed into a intermediate representation. Depending on the information needed for the comparison, the operation can be at the lexical or syntactical level. For example, comments and other uninteresting code fragments can be removed from the code, or abstract syntax tree can be built from code.

2. Code Comparison. Once the transformed code the actual comparison is done. It can be AST matching, line or metrics comparison.

Table 2 summarizes the different approaches used.

Halstead and Grier [18, 19] detect student plagiarism using statistical comparisons of style characteristics such as the use of operators, use of special symbols, frequency of occurrences of references to variables or the order in

Ref	Level	Transformed Code	Comparison Technique
[3]	Lexical	Substrings	String-Matching
[11]	Lexical	Normalized Strings	String-Matching
[2]	Lexical	Parameterized Strings	String-Matching
[5]	Syntactical	Metric Tuples	Discrete comparison
[27]	Syntactical	Metric Tuples	Euclidean distance
[7]	Syntactical	AST	Tree–Matching
[24]	Semantical	PDG	Relaxed subgraph matching
[25]	Semantical	PDG	Slicing

Table 2: Overview of the approaches for detecting duplication of code.

which procedures are called. Jankowitz [21] uses the static execution tree (the call graph) of a program to determine a fingerprint of the program.

Paul and Prakash [4] propose a regular language to identify programming patterns. Cloning can be detected if we assume that if two code fragments can be generated by the same patterns then they could be clones.

Johnson [3] applies a special heuristic, using constraints for the number of characters as well as the number of lines, to gather a number of lines into a *snip* of source code, to which he applies the fingerprint algorithm. Sif [22] is also based on the same idea. Johnson uses the identified duplication to understand change [28]. In [3], he also experimented with the abstraction of every sequence of alphanumeric characters to the letter 'i', i.e. a simpler and much less sensitive "parameterization" than what we are attempting here. Since he does a coarse grained interpretation of his results, little can be said save that he matches more code, as is expected.

Kontogiannis [6] evaluates the use of five data and control flow related metrics for identifying similar code fragments. The metrics are used as signatures for a code fragment. The technique supports change in the copied

code. However it is not language independent because it is based on Abstract Syntax Tree annotation.

dup [2] is a program that detects parameterized matches and generates reports on the found matches. This work is however mostly focused on the algorithmic aspects of detecting parameterized duplication and not on the application of the technique in an actual software maintenance and reengineering context.

Baxter et al. [7] transform source code into abstract syntax trees and detect clones and near miss clones among trees. They report similar code sequences and propose unifying macros to replace the found clones. Their approach requires, however, a full-fledged parser.

Kamiya et. al. [17] transform source code into tokens, removing identifier names and constants just like we have described in section 4. They remain largely language independent. As a comparison mechanism they employ suffix trees like dup [2]. They have shown their approach to be scalable to very large case studies.

Krinke [24] employs a program dependence graph to detect duplication through similarities of the data flow, reducing the dependence on syntactical similarity. This approach is however even more heavyweight than abstract syntax trees, since a detailed program dependence graph has to be built and compared.

Komondoor and Horwitz [25] also represent the code as a program dependence graph. Similar code fragments are extract using backwards slicing. This approach focuses on ideal clones which can subsequently be extracted and put into procedures automatically. The same drawback of high computational costs applies here as in [24].

In a recent comparison of some of detection techniques mentioned above, Bellon [29] has found that our approach could be categorized together with Baker [2] and Kamiya [17] as having high recall but low precision. Regarding the amount of returned clone candidates, our approach was closests to Baker's results. This seems to be attributable to an innate characteristic of both these techniques, namely to make the linebreaks in the source code a factor in the comparisons (which Kamiya does not). The same broad division of results along the borders of the fundamental techniques utilized was also visible for the other approaches.

5.1 Additional benefits of more sophisticated approaches

Detection approaches that use more heavyweight means benefit from basically two advantages:

- Better abstraction capabilities will detect duplication which is syntactically farther apart.
- More detailled information about the duplicated code provides more automation potential.

6 Conclusions and Future Work

We have presented a lightweight and language-independent technique to identify duplicated code. We have also demonstrated that more sophisticated approaches, such as parameterized matches [2], offer only small advantages over the lightweight approach based on simple string matching. We found that the *upper bound* limit of the missed duplicated code is at maximum %8 and in for sequence 10 lines less than 5% in average.

Detection of duplicated code is just one activity in the reengineering lifecycle [1]. It is a typical symptom of old age in legacy systems, and can be cured by refactoring operations [30, 31, 32]. We have applied the same principle of using lightweight techniques to develop Moose, a language-independent platform for reverse engineering and reengineering of object-oriented software [33, 34]. We have confirmed that simple, lightweight visualization techniques are useful not only for understanding duplicated code, but also for obtaining both coarse-grained and fine-grained views of complex software systems [35].

In the future, we would like to investigate how detection of code duplication could be productively integrated into a toolkit for reverse engineering and reengineering complex software systems.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation (SNF) and the Swiss Federal Office for Education and Science (BBW) for the projects "Framework-based Approach for Mastering Object-Oriented Software Evolution" (FAMOOS), ESPRIT Project 21975 / Swiss BBW Nr. 96.0015, "A Framework Approach to Composing Heterogeneous

Applications", Swiss National Science Foundation Project No. 20-53711.98, and "Meta-models and Tools for Evolution Towards Component Systems", Swiss National Science Foundation Project No. 20-61655.00.

References

[1] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2002.

[2] B. S. Baker, "A Program for Identifying Duplicated Code," *Computing Science and Statistics*, vol. 24, pp. 49–57, 1992.

[3] J. H. Johnson, "Substring matching for clone detection and change tracking," in *Proceedings of the International Conference on Software Maintence*, 1994, pp. 120–126.

[4] S. Paul and A. Prakash, "A framework for source code search using program patterns," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463–475, June 1994.

[5] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *International Conference on Software System Using Metrics*, 1996, pp. 244–253.

[6] K. Kontogiannis, "Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics," in *Proceedings Fourth Working Conference on Reverse Engineering*, I. Baxter, A. Quilici, and C. Verhoef, Eds. 1997, pp. 44 — 54, IEEE Computer Society.

[7] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of ICSM*. IEEE, 1998.

[8] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Metrics '99*, 1999, pp. 292–303.

[9] B. S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," in *Proceedings of the second IEEE Working Conference on Reverse Engineering (WCRE)*, July 1995, pp. 86–95.

[10] B. S. Baker, "Parameterized Pattern Matching by Boyer-Moore Type Algorithms," in *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan. 1995, pp. 541–550.

[11] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings ICSM '99 (International Conference on Software Maintenance)*, H. Yang and L. White, Eds. Sept. 1999, pp. 109–118, IEEE.

[12] G. G. Koni-N'sapu, "A scenario based approach for refactoring duplicated code in object oriented systems," Diploma thesis, University of Bern, June 2001.

[13] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl*, O'Reilly & Associates, Inc., 3rd edition, 2000.

[14] A. Gibbs and G. McIntyre, "The diagram: a method for comparing sequences. its use with amino acid and nucleotide sequences.," *Eur. J. Biochem.*, vol. 16, pp. 1–11, 1970.

[15] J. Helfman, "Dotplot Patterns: a Literal Look at Pattern Languages," *TAPOS*, vol. 2, no. 1, pp. 31–41, 1995.

[16] S. Wu and U. Manber, "Agrep — a fast approximate pattern matching tool," in *Proceedings of the Usenix Winter 1992 Technical Conference*, 1992, pp. 153–162.

[17] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 6, pp. 654–670, 2002.

[18] M. H. Halstead, *Elements of Software Science*, Elsevier North-Holland, 1977.

[19] S. Grier, "A Tool that Detects Plagiarism in PASCAL Programs," *SIGSCE Bulletin*, vol. 13, no. 1, 1981.

[20] N. H. Madhavji, "Compare: A Collusion Detector for PASCAL," *Techniques et Sciences Informatiques*, vol. 4, no. 6, pp. 489–498, Nov. 1985.

[21] H. T. Jankowitz, "Detecting Plagiarism in Student PASCAL Programs," *Computer Journal*, vol. 1, no. 31, pp. 1–8, 1988.

[22] U. Manber, "Finding Similar Files in a Large File System," in *Proceedings of the Winter Usenix Technical Conference*, 1994, pp. 1–10.

[23] R. Johnson and M. Palaniappan, "Metaflex: A flexible metaclass generator," in *Proceedings ECOOP '93*, O. Nierstrasz, Ed., Kaiserslautern, Germany, July 1993, vol. 707 of *LNCS*, pp. 503–528, Springer-Verlag.

[24] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings Eigth Working Conference on Reverse Engineering (WCRE'01)*. IEEE Computer Society, Oct. 2001, pp. 301–309.

[25] R. Komondoor and S. Horwitz, "Eliminating duplication in source code via procedure extraction," Tech. Rep. 1461, UW-Madison Dept. of Computer Sciences, Dec. 2002.

[26] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," in *Proceedings Seventh Working Conference on Reverse Engineering (WCRE'00)*, F. Balmas and K. Kontogiannis, Eds. IEEE Computer Society, Oct. 2000, pp. 98–107.

[27] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching for Clone and Concept Detection," *Journal of Automated Software Engineering*, vol. 3, pp. 77–108, 1996.

[28] J. H. Johnson, "Using textual redundancy to understand change," in *Proceedings of CASCON '95*, 1995, p. CD ROM.

[29] S. Bellon, "Vergleich von Techniken zur Erkennung duplizierten Quellcodes," M.S. thesis, Universität Stuttgart, Sept. 2002.

[30] E. Casais, "An incremental class reorganization approach," in *Proceedings ECOOP '92*, O. L. Madsen, Ed., Utrecht, the Netherlands, June 1992, vol. 615 of *LNCS*, pp. 114–132, Springer-Verlag.

[31] W. F. Opdyke, *Refactoring Object-Oriented Frameworks*, Ph.D. thesis, University of Illinois, 1992.

[32] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.

[33] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," in *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[34] S. Tichelaar, *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*, Ph.D. thesis, University of Berne, Dec. 2001.

[35] M. Lanza and S. Ducasse, "Polymetric views — a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–795, Sept. 2003.

Appendix A

These tables show the sizes of the case studies. Numbers shown like this *(100 KB)* are estimated.

6.1 Industrial Case Studies

ID	*Name*	*Language*	*Size*	*# Files*	*total LOC*	*effective LOC*
I1	Mail Sorting Controller	C++	4.65 MB	285	289436	95109
I2	Database Server	Smalltalk	6.83 MB	593	243755	142504
I4	Pipeline Simulation	C++	1.24 MB	49	43174	20772
I5	Microsoft Foundation Classes	C++	3.55 MB	245	144575	76589
I6	Payroll I	Cobol	2.94 MB	13	40515	19541
I7	Payroll II	Cobol	2.61 MB	336	35358	35358
I9	Forest Managment	C++	707.06 KB	63	24137	14222

6.2 Open Source Case Studies

ID	*Name*	*Language*	*Size*	*# Files/Classes*	*total LOC*	*effective*
O1	Agrep Info Retrieval	C	281.96 KB	22	13891	9789
O2	Aida Web Server	Smalltalk	500 KB	61	11175	8017
O3	Apache Web Server	C	2.69 MB	141	94874	5092
O4	Bison Compiler Generator	C	496.56 KB	54	20256	1059
O5	Flex Lexer Generator	C	470.44 KB	21	17549	1063
O6	GNU CC	C	13.4 MB	221	*(800000)*	*(4600*
O7	GNU JSP Implementation	Java	306.92 KB	77	9818	4442
O8	GNU diffutils	C	448.05 KB	24	16052	8266
O9	JBoss Application Server	Java	1.72 MB	403	62574	2497
O11	Patch	C	79.90 KB	6	3350	2251
O12	Refactoring Browser	Smalltalk	*(250 KB)*	180	17808	1469
O13	SDL	C++	1.94 MB	207	77116	4520
O14	Tomcat Servlet Container	Java	1.69 MB	267	53615	1968
O15	VW SAX Parser	Smalltalk	*(160 KB)*	60	3874	3138
O16	XEmacs (C Source)	C	7.41 MB	255	268695	1385
O17	XEmacs (Lisp Source)	Lisp	3.76 MB	227	105240	5973
O18	ZooLib	C++	2.51 MB	111	82662	4665

6.3 Academic Software Case Studies

ID	*Name*	*Language*	*Size*	*# Files/Classes*	*total LOC*	*effective LO*
A1	CodeCrawler	Smalltalk	*(150 KB)*	64	5794	4382
A2	Duploc	Smalltalk	1.5 MB	294	35176	20573
A3	Jun 3D Library	Smalltalk	5 MB	299	42151	48884
A4	Message Board	Python	265 KB	36	6709	6500
A5	Moose Reeng. Env.	Smalltalk	*(500 KB)*	105	12017	7742
A6	Sic Compiler Generator	Smalltalk	*(170 KB)*	58	8958	7558
A7	TGen Compiler Generator	Smalltalk	300 KB	92	5958	4205

Appendix B

The following tiny scanner in `perl` removes all comments from C/C++/Java programs. To be able to deal with special cases like comment delimiters within literal strings, strings are also recognized (and can be optionally transformed as well). The program additionally removes all white space from the source text and turns all characters to lowercase.

```
1   $inputFile  = $ARGV[0];
2   $outputFile = $ARGV[1];
3   open IN, $inputFile or die "Could not open '$inputFile': $!.\n";
4   $sourceText = join '', <IN>;
5   close IN;
6
7   $sourceText =~ s{
8                    (
9                      /\*.*?\*/  # recognize multiline comments
10                     |
11                     //.*?\n    # recognize C++-style comments
12                     |
13                      "          # recognize literal strings
14                       (
15                         \\.     # do not choke on "\""
16                        |
17                         [^\\]   # recognizes all unescaped characters
18                       )*?
19                      "
```

```
                    |
                     '          # recognize literal characters,
                      \\?       # be aware of '\n' etc.
                     .
                    '
                  )
                 }
                 {
                  treatDelimitedText($&)
                 }xgse;

$sourceText =~ s/[ \t\f\r]//g; # remove all whitespace except newlines

open OUT, ">$outputFile" or die "Could not open '$outputFile': $!.\n";
print OUT lc($sourceText);     # change all text to lowercase
close OUT;

sub treatDelimitedText {
  my ($delimitedText) = @_;
  # Comments: remove everything except for newlines
  if($delimitedText =~ m{^(/\*|//)}) {
    $delimitedText =~ s/[^\n]//g;
  }
  # Literal Strings/Characters: turn into spaces, leave but the delimiters
  if($delimitedText =~ m{^(\"|\')}) {
    $delimitedText = substr($delimitedText,0,1)
                   . ' 'x(length($delimitedText)-2)
                   . substr($delimitedText,0,1);
  }
  return $delimitedText
}
```

Appendix C

The following `perl` 5 regular expression implements a scanner with lookahead one non-whitespace character which recognizes identifiers in C code and transforms them into arbitrary strings. A list of all keywords of the

programming language is needed to distinguish identifiers. The input to this expression is expected to be free of comments or literal strings. Note that the expression does not match function definitions or function calls.

```
1   $sourceText =~ s/
2                     ([a-zA-Z_]\w*)   # recognizes alphanumeric strings
3                     (
4                      \s*?            # identifiers are followed by white space
5                      [               # or any of the characters from this list.
6                       \)\}\[\]\/\-   # Note that ( is not part of this list,
7                       +*!?=|&<>.,;:  # thus failing to match function
8                      ]               # definitions or function calls.
9                     )
10                  /
11                    replaceIfNotKeyword($1) # do not transform keywords
12                    .$2                     # do not loose lookahead
13                  /xeg;
```

The regular expressions to match other elements of program text, *e.g.*, constants of various types, are much simpler.