ISSN No. 0976-5697

Volume 4, No. 6, May 2013 (Special Issue)

International Journal of Advanced Research in Computer Science

REVIEW ARTICAL

Available Online at www.ijarcs.info

Detection Of Duplicate Code

Ms. Kanchan A. Jadhao
B.E. Final Year I.T
J.D.I.E.T. Yavatmal
Jadhav.kanchan196@gmail.com

Mr. Shubhendra R.Puri
B.E. Final Year
J.D.I.E.T. Yavatmal
purishubh@gmail.com

Ms.Nikita S.Varhade
B.E.final Year I.T
J.D.I.E.T.Yavatmal
nikitavarhade11@gmail.com

Prof. Ganesh B.Regulwar
Assistant Professor
J.D.I.E.T. Yavatmal
ganeshregulwar@gmail.com

Abstract: Task of managing duplicated or "cloned" code has occupied the minds of programmers for the past 50 years. During this time, researchers and practitioners have developed a variety of techniques for removing or avoiding it by employing functions, macros and other programming abstractions. Functional abstraction was designed into early programming languages, such as Fortran and Lisp. Object-oriented programming, originating with Simula-67, has provided further mechanisms for parameterized reuse to avoid duplication. Aspect-oriented programming has allowed cross-cutting duplication to be abstracted. Engineering practices like Refactoring and Extreme Programming have promoted specific methodologies of abstracting duplicated code. In the last decade, a multitude of tools have been developed (both in research and in industry) that help programmers semi-automatically find and refactor existing duplication into functions, macros and methods. Given this long-term commitment to programming abstractions as a solution use "duplicated code" and "cloned code" synonymously to mean two or more multi-line code fragments that are either identical or similar, particularly in their structure. Duplicated code, it stands to reason that there should be little duplication left in practice.

Keywords: Software maintenance, code duplication detection, code visualization

I. INTRODUCTION

Duplicated code is a phenomenon that occurs frequently in large systems. The reasons why programmers duplicate code are manifold and include the following reasons: (a) Making a copy of a code fragment is simpler and faster than writing the code from scratch. In addition, the fragment may already be tested so the introduction of a bug seems less likely. (b) Evaluating the performance of a programmer by the amount of code he or she produces gives a natural incentive for copying code. (c) Efficiency considerations may make the cost of a procedure call or method invocation seem too high a price. In industrial software development contexts, time pressure together with points (a) and (b) lead to plenty of opportunities for code duplication. Although code duplication can have its justifications, it is considered bad practice. Especially during maintenance.

If one repairs a bug in a system with duplicated code, all possible duplications of that bug must be checked. (b) Code duplication increases the size of the code, extending compile time and expanding the size of the executable. (c) Code duplication often indicates design problems like missing inheritance or missing procedural abstraction. In turn, such a lack of abstraction hampers the addition of functionality. Techniques and tools for detecting duplicated code are thus a highly desired commodity especially in the software maintenance community and research has proposed a number of approaches with promising results. However, the application of these techniques in an industrial context is hindered by one major obstacle: the need for parsing. Code duplication is one of the factors that severely complicates the maintenance and evolution of large software systems. Techniques for detecting duplicated code exist but rely mostly on parsers, technology that has proven to be brittle in the face of different languages and dialects. In this paper we show that is possible to circumvent this hindrance by applying a language independent and visual approach, i.e. a tool that requires no parsing, yet is able to detect a significant amount of code duplication. We validate our approach on a number of case studies, involving four different implementation languages and ranging from 256 K up to 13Mb of source code size.

II. FUNDAMENTS

This chapter's goal is to introduce some of the idioms and principles that guide now a days the object oriented design and programming.

III. OBJECT-ORIENTED PROGRAMING

Object-oriented methods provide a set of techniques for analyzing, decomposing, and modularizing software system architectures. The object-oriented programming is concerned with implementation issues and is highly dependent on the object-oriented programming languages.
The main mechanisms provided by the modern object-oriented programming languages are:

a. abstract data types (classes)
b. encapsulation
c. inheritance

d. polymorphism

Encapsulation is basically described as hiding data. Objects generally do not expose their internal data members to the outside world (that is, their visibility is protected or private). But encapsulation refers to more than hiding data. The advantage of using encapsulation is that more we make our objects responsible for their own behaviors, the less the controlling programs have to be responsible for. Encapsulation makes changes to an objects internal behavior transparent to other objects. Encapsulation helps to prevent unwanted side effects. With encapsulation the data structure of a class is hidden behind an interface of operations.

a. ***Inheritance:*** is another vital mechanism of object-oriented programming. Instead of defining every time new types from scratch, we can use types (classes) that already exist and specialize them. This is the support for the is-a relationship: having one class be a special kind of another class. The base class (called the super class) can be extended by any number of new classes and this is how class hierarchies appear.

b. ***Polymorphism:*** is the ability of related objects to implement methods that are specialized to their type. We are able to refer to different derivations of a class in the same way, but getting the behavior appropriate to the derived class being referred to. This way, it offers basis for flexible architectures and designs. The high-level logic is defined in terms of abstract interfaces and relies on the specific implementation provided by the subclasses. What we apparently refer are objects with one type of reference that is an abstract class type. However, what we are actually referring to are specific instances of classes derived from their abstract classes. The subclasses can be added without changing high-level logic. Objects of the subclasses can be dynamically interchanged without affecting their clients.

IV. CODE DUPLICATION OVERVIEW

In this section, we will present the overall percentages of duplication which we extracted from the reports produced by our tool .We take these numbers to be nothing more than very general indicators of duplication occurring in a system. We will not go into a more detailed analysis of the reports, since our aim in this section is Programmers employ functions, macros, classes, aspects, templates, and other programming abstractions to reduce duplication. The identical sections of the clones become the body of the abstraction’s definition, and the differences become parameters. However, abstractions can be costly, and it is often in a programmer’s best interest to leave code duplicated instead. Specifically, we have identified the following general *costs of abstraction* that lead programmers to duplicate code (supported by a literature survey, programmer interviews, and our own analysis). These costs apply to any abstraction mechanism based on named, parameterized definitions and uses, regardless language.

V. AVERAGE PERCENTAGE OF DUPLICATION

The following table presents the average percentage of duplication per file. We also include the percentage in terms of entire code (i.e. files including comments) so that readers can have their own ideas about the relevance of the duplication detection. The third line shows the number of files that effectively contain duplicated code under the constraints we fixed Note that inferior percentages for the entire code is normal because comments and white space can make up for a lot of lines. Average percentage of duplication found per file Case gcc Database Payroll Mess. B. effective LOC 8.7% 36.4% 59.3% 29.4% entire LOC 5.9% 23.3% 25.4% 17.4% # of files 143 464 13 24 with duplication Total # of Files 170 593 13 36 The quite high average percentage found for the two industrial case studies (Cobol payroll system and Smalltalk database server) is not totally surprising considering fact that these were given to us because it was suspected that they contained These thresholds come from our experiences with the case studies a lot of duplication. Nevertheless we were astounded by their overall duplication ratio. The web message board system shows some duplication elements that are result from evolutionary clones, since the system was given to us as a snapshot in the middle of an extension, thus containing old as well as new code side by side. The gcc source code has the lowest ratio. This is not surprising because gcc is known to be software of a good quality. Now we refine our analysis by looking at the duplication percentage per file. We present the payroll system and gcc because they cover the extremes in the range of our case studies. Note that the tables in Figure 1 display the files containing the effective duplication.

Table 1.Average percentage of duplication found per file

Case	**gcc**	**Database**	**payroll**	**Mess.B.**
effective LOC	8.7%	36.4%	59.3%	29.4%
entire LOC	5.9%	23.3%	25.4%	17.4%
# of files with duplication	143	464	13	24
Total # of Files	170	593	13	36

a. ***Percentage per File**: **the Payroll Case***: For the payroll system, the overview immediately identifies three main groups according to the degree of duplication: (a) few duplication (around 5% in file F), (2) some duplication (from 25% to 50% in files A, B, D, E and J) and (3) mostly duplicated (up to 70% in files C, G, H, I, K, L and M).

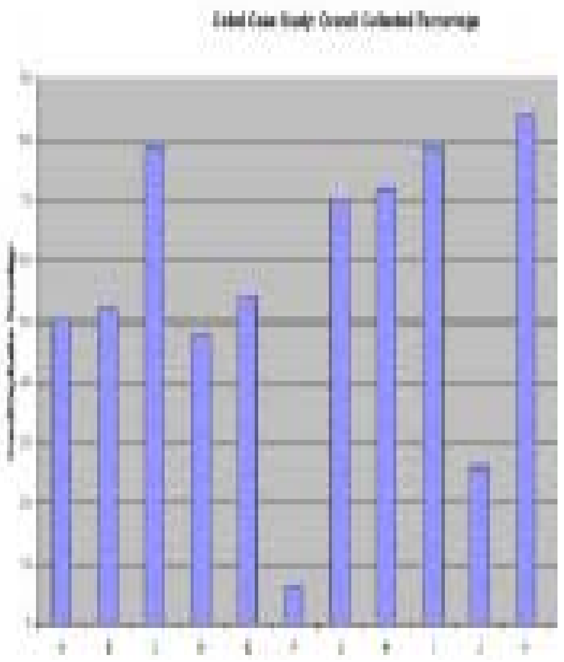
Figure1.Duplication Percentage per Files in the payroll case study.

b. Percentage per File: the gcc Case: Even if the average percentage showed that gcc has the lowest percentage of duplication, looking at the percentage per file gives another view. We see that two files have more than 60% of duplication, that 6 files have more than 50% of duplication and that a number of files have more than 20% of duplication. The data from the reports that the analysis of this section was made with also serves the software maintainer in the process of eliminating duplication. What we want to do in the next section is to look at line-based comparison data from the angle of its representation in scatter-plots.

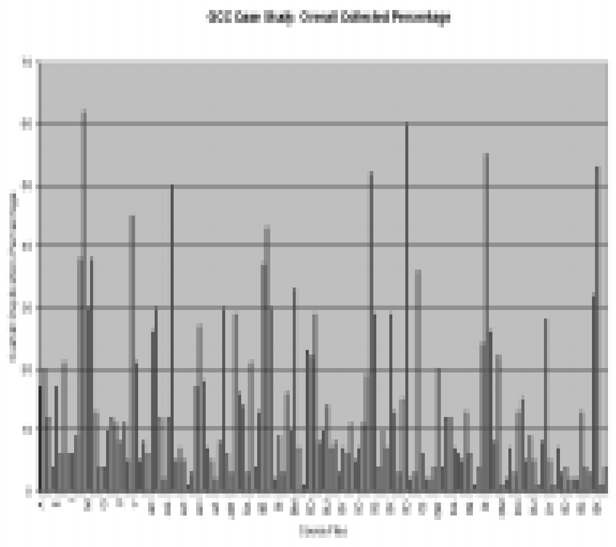
Figure2.Percentage per File: the gcc Case.

VI. ALGORITHM'S PRINCIPLES

The approach chosen for this tool is an enhanced scatter-plot approach. scatter-plot approach is not new to software research and it is based mainly on:

a. bringing the code to a brute state (non-indented, comments off)
b. building a matrix that will store the results of matching between the lines of code
c. populating it, by marking every match
d. presenting it to the specialist, for further visual studying

The enhancement we propose is to try to unite copied sequence of code that are close enough to each other and merging them into a cluster of code duplication. The tool will report a list of clusters (chains). When introducing code clones, programmers often change white spacing (blanks, tabs, newlines) and comments, which will disable recognition based purely on strings. In order to combat this problem, the presented tool transforms the code lines by "cleaning" the code. Clone detection is a process in which the input is a list of source files (or method bodies) and the output is a list of duplication chains. The entire process of our string comparison clone detecting technique consists of the following steps.

A. Code cleaning:

After reading the source code lines, the first thing to do is bringing the code to a raw state, in order to avoid situations where identical lines which are differently indented or having comments added are not reported as duplicates**.**

a. striping the comments (optionally, only Java and C,C++)
b. removing any whitespaces (including nice indentation)
c. removing noise (specified in a file)

Lines of code containing only a keyword (else) or some other syntactic element (an open or a closed brace), which can be considered as less relevant to the duplication issue.

B. Building the matrix:

The lines of code the cleaning this phase (further referred to as relevant lines) will be stored in the exact order they were read: all the relevant lines of the first file, followed by the ones of the second file, and so on. The next step consists of building a two-dimensional matrix NxN, where N is the total number of relevant lines in the system. An element of the matrix Element[i,j] will store the result of matching the relevant lines i and j.This way, every line will be compared with every other line in the system (exhaustive approach). Only the hits of the matching will be marked in the matrix.

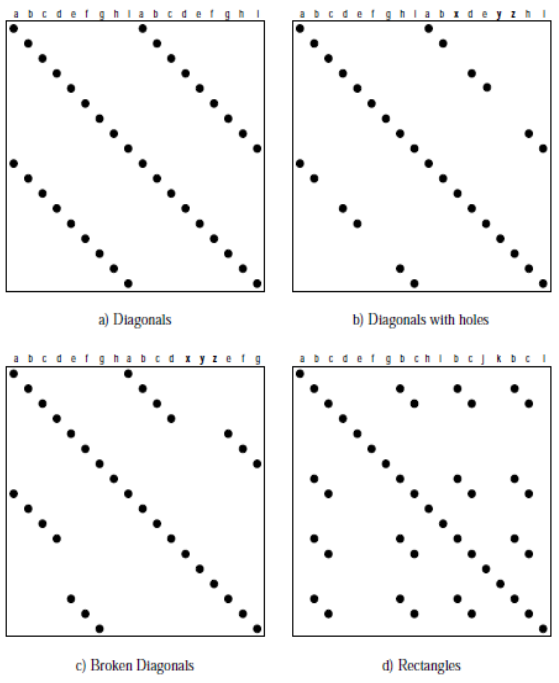

Figure 3.Four steps for Building the matrix.

VII. USER STUDY

We conducted a user study to compare the use of Code link with programming abstractions. Our hypotheses were that programmers would be able to link clones with Code link in much less time than it would take to abstract the clones, and that Code link would provide programmers with comparable benefits after linking the code. We paid 13 students from U.C. Berkeley to participate in the study. Subjects had a diverse range of programming skill, ranging from graduate students in Computer Science to introductory-level undergraduates. Subjects performed their programming tasks in the Scheme programming language since functional abstractions in Scheme are expressively powerful and well-understood by students at Berkeley, thus mitigating biases from language-specific abstraction costs. We expect the results to transfer to functions, macros and methods in other languages as well. We used a within-subjects experimental design. With both functional abstraction and Code link, subjects were asked to perform a set of programming tasks:

a. To abstract or link two short pieces of cloned code.
b. To perform a modification task requiring new code to be added to both clones or instances
c. To perform a modification task requiring new differences between each clone or instance.

We believe these programming tasks the tradeoffs in editing duplicated/abstracted code. Although both sets of programming tasks followed the general sequence given above, the specific tasks and code were very different for each technique to eliminate learning effects. The pairing between techniques and task-sets and the ordering of techniques used were fully counterbalanced to eliminate ordering, learning and task biases. Before performing each set of programming tasks, subjects completed a short (5-10 minute) tutorial to teach them about the technique (functional abstraction or code link, depending on the condition) and what was expected of them on the tasks. The tutorial walked them through the three types of programming tasks, with very simple code and modifications. Subjects filled out a questionnaire after each experimental task-set to assess the particular technique paired with that task-set. The entire study lasted between 30 and 90 minutes. The programming tasks were recorded with a screen-capture program, and audio was captured and merged into the video to facilitate data analysis. Subjects did not test their code; they were rather instructed to stop when they thought their code would work.

A. *Evaluation metrics:*

We recorded two dependent variables: the **time** it took subjects to link or functionally abstract the code (Step 1 in each set of programming tasks), and the **ratings** they gave each technique on the post-task questionnaires. Abstraction/ link time was measured from the subject's first key press after reading the task instructions to the last key press before flipping to the next task's instructions. On the post-programming questionnaires, subjects rated each technique along the following five metrics: maintainability, understandability, changeability, editing speed, and editing effort; reported on a 7-point semantic differential scale. Each question asked subjects how the technique they used (functional abstraction or Codelink) helped or hindered them on the programming tasks, as compared to editing the duplicated code directly. Finally, the experimenter asked subjects the following question verbally: "If you had the Code link tool in your editor programming environment, and the other programmers on your programming project had it too, how likely is it that you would use it in your own programming work?" The responses were classified into three groups: probably or definitely wouldn't use Code link, not sure, and probably or definitely would use Code link. Because this question was only incorporated into the study after the first three subjects had been run, we only received 10 responses instead of 13.

VIII. EVALUATION OF THE TOOL

The graphical user interface offers a simple, yet powerful access to the duplication chains detecting engine. The all-in-one-window integrated workspace is composed of:

a. control panel
b. parameters panel
c. results panel (a list of found chains)
d. visualization panel, for visual analysis of the duplicated code involved in a chain
e. • status bar

In order to analyze a project, first you have to set the starting path (current directory) where the source files of the project are located. Then, you can modify the searching parameters and hit the Search button. The status bar contains a progress bar, visible only during a search process. After the searching is over, if any duplication chains were found, they will be shown in a list of chains which can be sorted by any of: entity's name, index to the first or the last line of code in the chain, length, type etc. If you would like to save the results for subsequent analysis, you can ask to generate a report (the Save Results button), which stores the list of results in a specified file, with respect to the current sorting of the list. I order to validate the duplication chains or to examine the results in a visual manner, a mouse click on any of the items in the results list will display the contents of the 2 files involved in that duplication in the Duplicate Viewer panel, with the replicated code highlighted in yellow.

IX. BENEFITS OF THE APPROACH

As stated in the Introduction, the major benefits of a slicing-based approach to clone detection are the ability to non-contiguous, reordered, and intertwined clones, and the likelihood that the clones that are found are good candidates for extraction. These benefits, discussed in more detail below, arise mainly because slicing is based on the PDG, which provides an abstraction that ignores arbitrary sequencing choices made by the programmer, and instead captures the important dependences among program components. In contrast, most previous approaches to clone detection used the program text, its control- graph, or its abstract-syntax tree, all of which are more closely tied to the (sometimes irrelevant) lexical structure. Finding non-contiguous, reordered, and intertwined clones: One example of non-contiguous clones indented by our tool was given in Figure 4. By running a preliminary implementation of the proposed tool on some real pro-grams, we have observed that non-contiguous clones that are good candidates for extraction (like the ones in Figure 1) occur frequently (see Section 3 for further discussion). Therefore, the fact that our approach can send such clones is a significant advantage over most previous approaches to clone detection. Non-contiguous clones are a kind of near duplication. from the Unix utility sort is given in Figure 4. In this example, one clone is indicated by \++" signs while the other clone is indicated by \xx"signs. The clones take a character pointer (a/b) and advance the pointer past all blank characters, also setting a temporary variable (tmpa/tmpb) to point to therst non-blank character. The external component of each clone is an if predicate that uses the temporary. The predicates were the starting points of the slices used to and the two clones the second one {the second-to-last line of code in the figure { occurs 43 lines further down in the code).

```
++ tmpa = UCHAR(*a),
xx tmpb = UCHAR(*b);
++ while (blanks[tmpa])
++ tmpa = UCHAR(*++a);
xx while (blanks[tmpb])
xx tmpb = UCHAR(*++b);
++ if (tmpa == '-') {
tmpa = UCHAR(*++a);
...
}
xx else if (tmpb == '-') {
if (...UCHAR(*++b)...)
```

Figure 4.An intertwined clone pair from sort**.**

Although the study results are promising, there are anumber of obstacles to be overcome before Codelink is aviable option in real-world projects. The LCS algorithm used in the prototype, although adequate for the user study, has two shortcomings: it takes O(nk) time (for k clones of size n), and does not always report the most intuitive set ofdifferences between any two code fragments. (Some of the issues are described by Heckel [22]). We are developing better differencing algorithm that uses interactive syntactic information (provided by the Harmon is framework) to derive differences that more closely correspond to the way humans view duplicated code, with a much faster running time. We are also revising the incremental re-differencing algorithm, and developing a mechanism to allow users to give feedback and fine-tune the types of differences reported by the algorithm. Apart from differencing, we can also experiment with when to invoke differencing engine and link the clones. While the current implementation requires the user to select all clones and click “Link Selections,” even this step could be performed automatically, either as a result of the user copying and pasting code or via the output of a third-party clone-finding tool, such as the ones mentioned in Section 1.

If programmers are able to link many clones simultaneously across the breadth of a project, an “overview” window or pane that visualizes all linked clones as the programmer edits one of them would be necessary. We would also like to make our link meta-data resilient to file modifications made by third-party tools. Lastly, we notice that there are often higher-level patterns to clones (like consistent variable renaming) for which Linked Editing may be able to infer and provide automated editing support. We are working on many of these issues and plan to release a more robust version of Code link to the public as an open-source software project in the future. This will allow us to get real-world usage data to verify that Linked Editing can scale to real programs, and that programmers would use it in their real work. Finally, we are working to extend the general technique of Linked Editing to support documents in non programming domains, such as spreadsheets, web sites, form letters, graphic charts, and music scores. Although these documents frequently contain duplicated content, their authoring environments provide impoverished or nonexistent abstraction facilities, and are frequently used by non-programmers. We feel that Linked Editing could provide a substantial benefit to these domains.

X. CONCLUSION

We described Linked Editing, a technique that augments a text editor to provide programmers with a lightweight mechanism to read, write, and edit patterns of duplicated code in an abstract way. We implemented a prototype of Linked Editing named Code link, and compared it to functional abstraction in a user study. The study found that Linked Editing can provide the same benefits as functional abstractions with drastically less work. Most subjects said they would use a tool like Code link in their real-life work. These results indicate that Linked Editing would be likely to be used in practice by developers, and would be powerful enough to alleviate the issues of duplicated code in many situations. Software developers continuously deal with reading, writing, and maintaining programs that are infused with duplicated code because functions, macros and other programming abstractions don't adequately support their needs. An improvement to this situation would greatly benefit the state of software development at large.

XI. REFERENCES

[1]. Brenda S. Baker. A Program for Identifying Duplicated Code.

[2]. J. H. Johnson. Substring Matching for Clone Detection and Change Tracking.

[3]. K. Kontogiannis. Evaluation on the Detection of Programming Patterns Using Software Metrics.

[4]. J. H. Johnson. Substring Matching for Clone Detection and Change Tracking.

[5]. B. S. Baker. On finding duplication and near-duplication in large software systems.

[6]. S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code.

[7]. J. H. Johnson. Substring matching for clone detection and change tracking.

[8]. J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics.

[9]. Marcelo Sant Anna, Lorraine Bier. Clone Detection Using Abstract Syntax Trees.

[10]. Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Automatic detection of function clones in a software system using metrics