Volume 5, Issue 12, December 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Detection of Cloned Code to Improve Existing Software

Ganesh B. Regulwar
Assistant Professor, CSE Dept
BNCOE, Pusad
Maharashtra, India

Dr. R. M. Tugnayat
Principal
SSPACE, Wardha
Maharashtra, India

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: Duplicate code; reuse to avoid duplication; abstracting duplicated code; Refactoring; techniques for removing or avoiding.

I. INTRODUCTION

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 Kb up to 13Mb of source code size.

Task of managing duplicated or "cloned" code1 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.

II. LITERATURE OF VIEW

What Is Code Duplication Detection

Duplicated code is a phenomenon that occurs frequently in large systems. The reasons why programmers duplicate code are manifold and include the following reasons:

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.Evaluating the performance of a programmer by the amount of code he or she produces gives a natural incentive for copying code. Efficiency considerations may make the cost of a procedure call or method invocation seems too high a price. In industrial software development contexts, time pressure together lead to plenty of opportunities for code duplication. Although code duplication can have its justifications, it is considered bad practice.

© 2015, IJARCSSE All Rights Reserved

Especially during unjustified duplicated code gives rise to severe problems:

If one repairs a bug in a system with duplicated code, all possible duplications of that bug must be checked.Code duplication increases the size of the code, extending compile time and expanding the size of the executable.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. This is clearly stated in the following quote:

"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 and language ambiguities"

The Duplicated Code Problem

We wish to eliminate the problems of duplicated code, rather than the code itself. Thus, we must understand how duplicated code impedes the task of programming. We interviewed three programmers (computer science graduate students at U.C. Berkeley), and combined the results with a survey of the literature and our own analysis to identify the following four problems with duplicated code:

Verbosity: Redundant cloned code creates clutter and obscures meaningful information, making code difficult to understand.

Tedious, repetitive editing: Edits made to one clone must often be made to its copies. Thus, a single modification often requires many edits, making sustained modification unwieldy.

Lost clones: Although edits to one clone must often be made to other clones as well, there is no way of getting to those other clones from the first, or of even realizing that the others exist. Missed edits lead to inconsistent code.

Unobservable inconsistency: Even if programmers find all clones and edit each one, it is impossible to verify that the edits have been made consistently that the common regions are identical, and the differences are retained without manually comparing each clone's body, word-byword, and hoping that no important details were missed. These problems prevalence of duplicated code, define the duplicated code problem.

Why Programmers Duplicate Code

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 .These costs apply to any abstraction mechanism based on named, parameterized definitions and uses, regardless of the language.

Too much work to create: In order to create a new programming abstraction from duplicated code, the programmer has to analyze the clones' similarities and differences, research their uses in the context of the program, and design a name and sequence of named parameters that account for present and future instantiations and represent a meaningful "design concept" in the system. This research and reasoning is thought-intensive and time-consuming.

Too much overhead after creation: Each new programming abstraction adds textual and cognitive overhead the abstraction's interface must be declared, maintained, and kept consistent, and the program logic (now decoupled) must be traced through additional interfaces and locations to be understood and managed.

Too hard to change: It is hard to modify the structure of highly-abstracted code. Doing so requires changing abstraction definitions and all of their uses, and often necessitates reordering inheritance hierarchies and other restructuring, requiring a new round of testing to ensure correctness. Programmers may duplicate code instead of restructuring existing abstractions, or in order to reduce the risk of restructuring in the future.Too hard to understand: Some instances of duplicated code are particularly difficult to abstract cleanly, e.g. because they have a complex set of differences to parameterize or do not represent a clear design concept in the system. Furthermore, abstractions themselves are cognitively difficult. To quote Green & Blackwell: "Thinking in abstract terms is difficult: it comes late in children, it comes late to adults as they learn a new domain of knowledge, and it comes late within any given discipline."

III. RELATED WORK

The analysis of code to identify copy and paste and plagiarism is broad. Various techniques are used: structural comparison using pattern matching, metrics or statistical analysis of the code, code fingerprints 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 which procedures are called static execution tree (the call graph) of a program to determine a fingerprint of the program. In a regular language is proposed 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 beclones. Johnson uses a specific heuristic, using constraints for the number of characters as well as the number of lines, to gather a number of lines a snip of source code on which he applies the fingerprint algorithm. However no graphical support is provided and the reports only present an overall similarity percentage between two files. Evaluates the use of five data and control flow related metrics for identifying similar code fragments. The metrics are used as signatures for a

© 2015, IJARCSSE All Rights Reserved

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. The visual display used in DUPLOC is not new, DOTPLOT uses the same principle. DOTPLOT has been used to compare source code, but also filenames in a file system and literary and technical texts. However it does not support source code browsing and the report facilities. DUPLICATE 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. In particular code browsing not supported. The tool of transforms source code into abstract syntax trees and detects clones and near miss clones among trees. It reports similar code sequences and proposes unifying macros to replace the found clones. Their approach requires, however, a full-fledged parser.

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 Codelink 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:

(1) To abstract or link two short pieces of cloned code.
(2) To perform a modification task requiring new code to be added to both clones or instances.
(3) To perform a modification task requiring new differences between each clone or Instances.

We believe these programming tasks illustrate 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 was 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.

Each technique on the post-task questionnaires. Abstraction/link time was measured from the subject's first keypress 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. With these questions, we judged how each technique helped or hindered programmers after the initial abstraction or linking of code. Finally, the experimenter asked subjects the following question verbally: "If you had the Codelink tool in your editor or 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 Codelink, not sure, and probably or definitely would use Codelink.

IV. SYSTEM STUDY

Code Duplication Overview

In this section, we 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 only to prove that our approach detects a significant amount of duplication.

Constraining the Results: The results we present here have been obtained with the following constraints: First, to remove accidental duplication of small fragments, we limited the detection to sequences of lines having 10 or more lines. Second, to avoid missing duplicated sequences with changed parts, we allowed holes in the copied sequences up to 20%4 of the total length of the sequence. Third, since reengineers are looking for the duplication of functional code elements, we chose to present the percentage of duplication in terms of effective lines of code. This means that we computed the percentage over the set of lines from which comments and white space have been removed (see section 2.1). This way we minimize the impact of comments in the percentage computation. As a consequence, a file can be integrally copied into a second, but if this second file contains a lot of comments the percentage will not reflect this situation.

© 2015, IJARCSSE All Rights Reserved

Overview of the Duplication

Having a global percentage of duplication per application is the first indication of the state of an application.

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 an inferior percentage for the entire code is normal because comments and white space can make up for a lot of lines.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 4These 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 Figures 3 and 4 only display the files containing the effective duplication. Percentage per File: the Payroll Case. For the payroll system, the overview (Figure 1) 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).

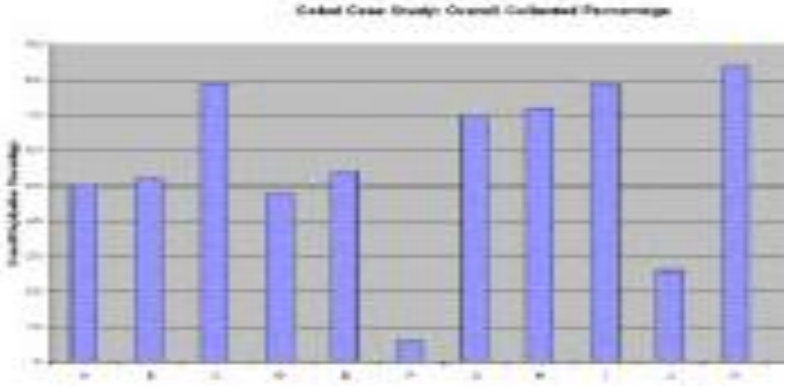


Figure 1.Duplication Percentage per Files in the payroll case study.

V. REQUIREMENT ANALYSIS

A Language Independent Approach

As we have stressed with the introductory quote, language dependency is a big obstacle when it comes to the practical applicability of duplication detection. We have thus chosen to employ a technique that is as simple as possible and prove that it is effective in finding duplication. In this section, we detail the principles behind our approach under the three aspects algorithms used to compute the comparisons data, visualization of the comparison data, and pattern matching to condense the data. Figure 1 shows an overview of the steps that take us from source code to duplication data.

Algorithmic Aspects

Clone detection is always a two-step process. First, source code is transformed into an internal format. Second, a more or less sophisticated comparison algorithm then performed on the internal data. In our case, the code is only slightly transformed using string manipulation operations. To compare the transformed lines, we use basic string matching.

Source Code Transformation

Two decisions must be made: the nature of the transformation and the size of the source code fragment that will be the entity of the incidental comparison. We choose one line of source code as code fragment entity on which we base our algorithm. The choice is on the one hand motivated by the consideration that the important copy and paste performed by programmers include one or more lines, and on the other hand that preprocessing can be kept simple. To start language independent, we refrain from code transformation to more abstract formats like AST's which have to employ parsing, or parameterized strings [1] which need at least a lexer. The transformation we apply to a code fragment is minimal and stays in the realm of string manipulation: We remove comments and all white space until we get a condensed form of the line.1 As an example, the C line is condensed to

```
if( code &pcObjType ) f /* print type */
```

is condensed to

```
if(code&pcObjType)
```

As a consequence, the code reader which does the transformation is adapted to any new language in a few minutes. The UNIX diff utility uses the same condensation technique upon request. The transformation reduces the entire file to an ordered collection of effective lines (see Figure 2) that will be compared against itself and line collections from other files.

© 2015, IJARCSSE All Rights Reserved

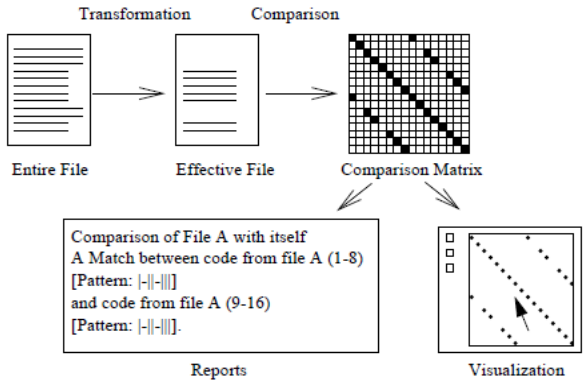


Figure 2.Overview of the approach.

Comparison Algorithm

Since we do not know what to look for, we cannot apply grep-like pattern matching algorithm but have to compare every entity (transformed source line) with every other entity. The comparison of two entities is done by string matching. The result is a boolean true for an exact match and a false otherwise. This value is stored in a matrix (see Figure 1), taking the coordinates that the two compared entities have in their respective ordered collections as the matrix coordinates for the comparison result. Note that after the comparison process, the matrix only contains matches of individual lines and not yet of whole sequences. They will have to be extracted from the matrix in a separate pass.

Optimization

The search space spanned by an input of n lines is quite uncomfortable (n2). We thus reduce it by preprocessing the transformed lines a second time: the lines are hashed into B buckets. The string matching is then applied on all possible pairs of one hash bucket. Equal lines have the same hash value and are thus thrown into the same hash bucket, so no false negatives occur. This procedure cuts the processing time by the factor B optimization is used.

Selecting the Case Studies

The goal of our case studies is to stress the language independent aspects and to prove the potential of our approach. In order to choose the case studies we took four criteria into account first the implementation language, second the potential of duplication, third the size of the system and fourth the possibility of reproduction of the experiment by other researchers.

Implementation Languages: We selected languages that have clearly different syntaxes: C, Smalltalk, Python and Cobol. Smalltalk is well known to have a simple and uniform, keyword based syntax. The syntax of Python is based on indentation which replaces block delimiters. In the overly verbose Cobol syntax, line numbers exist and identifiers that are attached at the end of each line. It is obvious that writing a parser for these diverse languages would be a entire new endeavor in each case.

Potential Duplication: We used case studies from different sources to maximize the potential range of the duplication. We took (a) two industrial case studies for which it was known that they contained a lot of duplication, (b) one case study where the duplication of code was suspected to be low, and (c) a small application from the public domain for 2DUPLOC is available under a GNU license at http://www.iam.unibe.ch/_rieger/duploc/.Which we had no knowledge about the duplication situation.

Size: The scalability of our approach has to be considered under two aspects: First, does it scale given the size of the source code? Second, does it scale given the amount of duplication that is found? To prove that our approach is scalable under the first aspect, we took the full GNU gcc source code whose size is 13Mb. That our approach also scales regarding the second aspect was proved when one of the industrial applications happened to contain an enormous amount of duplication.

Reproducibility of the Experiment: We chose one well-known and freely available case study: the Free Software Foundation C compiler gcc 3 to make our experiments reproducible by others.

The Case Study Material: The chosen case studies are: the implementation files of the GNU gcc source (written in C), a web-based message board (Python), parts of a payroll application (Cobol) and a database server (Smalltalk). Initial Set-Up: We have performed the case studies according to this procedure:

(1) We took source code about which we did not know anything.
(2) Since the code was written in a number of different languages, we had to adapt our tool to the language at hand so it could normalize the lines (white space and comment removal).
(3) We ran the tool off-line to produce the reports.
(4) We extracted overall duplication percentages from the reports and analyzed them.
(5) We browsed the visual representations of the matrices to evaluate our findings.

VI. IMPLEMENTATION DETAILS

Technical Remarks

We add some technical remarks about the tool we implemented.

© 2015, IJARCSSE All Rights Reserved

Scalability: The scalability issue is a crucial question when striving for applicability in the industrial context where system sizes of hundreds of KLOCs are normal. The table below summarizes some performance statistics gathered during our case studies. We have run the tool on a single G3 MacIntosh with 230Mhz and 100Mb RAM. It checked for all matching code sequences which where longer than 10 matching lines (including holes). Note that the task of computing the comparisons of a set of files can be easily parallelized.

Evolutionary Change

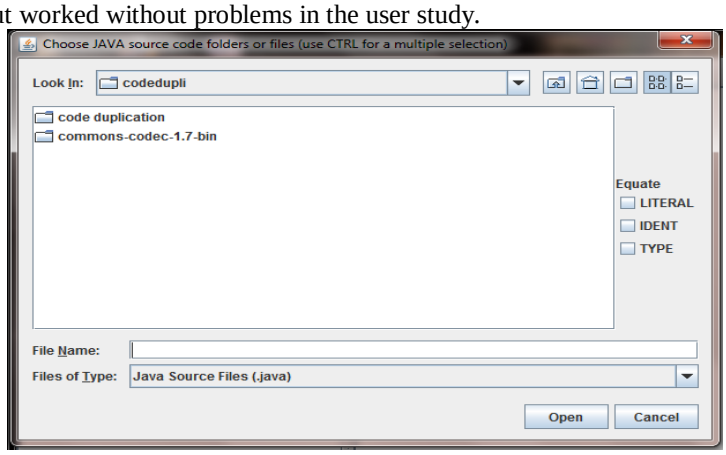
Knowing that the message board code contained old and new versions, we can observe how code evolution appears in scatter-plots. Most of the changes that were applied to the code consisted in adding lines. This shows up as broken diagonals that are progressively shifted to the right. In the middle of the diagram, however, we see a down-shift of the diagonal, telling us that a chunk of code has been removed. Since one matrix coordinate corresponds to one source line, we are able to estimate the size of the changed code chunks easily, which facilitates understanding further. Note that the same information could be obtained using the diff tool, but it is only via the image that the user understands the changes quickly. Note that See soft that interactively displays line oriented statistics like the age of the line and its programmer is a better suited tool for this kind of analysis.

Additional Comments about the Visualization

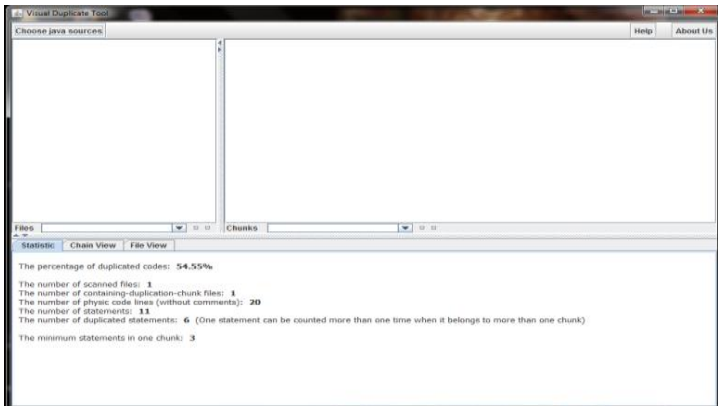
We experienced that the performance of the current implementation tool is sufficient for systems sizes below 1MLOC. We think that in a maintenance project, duplication data is something that does not change frequently and can be computed once over night and then be interpreted afterwards Should performance turn out to be a critical factor, a re-implementation of the tool in C++ would certainly amend the problem.

Implementation Work

Codelink is implemented within Harmonia-Mode anXEmacs extension that provides interactive program analyses via the Harmonia interactive language-analysis framework. It can be applied to many programming languages, thanks in part to the language independence of the Harmonia framework. A thorough user-centered, iterative design process was used to design and verify the usability of the user-interface prior to the study described. Our difference analysis uses the dynamic programming version of the longest-common-subsequence (LCS) algorithm operating on subsections of lexical units in the program obtained by splitting each token at "word" boundaries-dashes, underscores, whitespace characters, etc. The sequence of tokens returned by this LCS algorithm became the blue "common" regions, and all other tokens became the yellow regions. When an edit is made to a single clone, an incremental version of the difference algorithm is run. It isolates the token subsections in which the edit occurred, and then re-runs the original difference algorithm in the smallest yellow region that fully encompasses this edited region. This method of re-differencing changed regions is not optimal in all situations, but worked without problems in the user study.

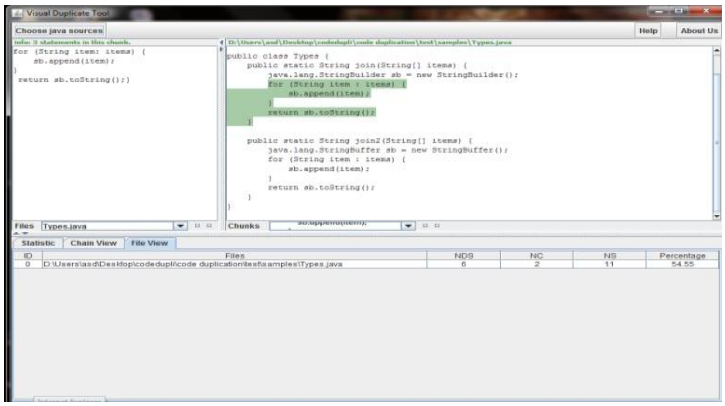


Screenshot 1.Selection of File



Screenshot 2.Statistic View

© 2015, IJARCSSE All Rights Reserved



Screenshot 3.File View

VII. CONCLUSION

In this paper, we presented a language independent approach for detecting duplicated code. The approach is based on (1) simple line-based string matching, (2) visual presentation of the duplicated code and (3) detailed textual reports from which overview data can be synthesized. We presented results from a number of case studies and we showed that we can easily identify (1) duplicated code between several files, (2) within the same file, (3) cloned files and (4) evolution files. We claim that the data that was generated is useful information for practical software maintenance and reengineering tasks.

Moreover, the results we found were in certain instances beyond our expectations. We are referring in particular to the two industrial case studies. The provider of the source code suspected code duplication in the system, but we found much more than expected. The high amount of duplication, especially in the database server case study, suggests that the simple algorithm does not miss too much of the duplication that is actually going on in the system. Evaluating the impact of a parameterized match is however one of our future goals.

VIII. FUTURE WORK

We plan to (a) Experiment with different algorithms for fuzzy matching, (b) Evaluate the impact of a parameterized comparison algorithm on this approach. We want to qualify how much of duplication we are missing and compare the pre-processing time, space trade-off and languages independent lose between the two approaches. (c) Evaluate the gain in time that Sif as a first filter of similar files can provide. (d) Develop a methodology for finding essential duplicated code in a system. This includes especially the fine tuning of the report facility. We will work in close contact with industrial software maintainers. (e) Investigate if textual comparison is a useful approach for exploring the structure of data other than source code, like for example program execution traces, where it could be interesting to find recurring patterns of execution sequences.

REFERENCES

[1] Brenda S. Baker, A Program for Identifying Duplicated Code Computer Science and Statistics: Proc. Symp. On the Interface, page 49--57. (March 1992).

[2] J. H. Johnson, Substring Matching for Clone Detection and Change Tracking Software Engineering Laboratory September, 1994 “Proceedings of the International Conference on Software Maintenance (ICSM),” Victoria, British Columbia, September 19–23, 1994, (pp. 120–126), Print ISBN: 0-8186-6330-8

[3] K. Kontogiannis, Evaluation on the Detection of Programming Patterns Using Software Metrics, Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on 6-8 Oct 1997(pp 44 – 54) Print ISBN**:** 0-8186-8162-4

[4] B. S. Baker, On finding duplication and near-duplication in large software systems, Reverse Engineering, 1995., Proceedings of 2nd Working Conference on 14-16 Jul 1995 (pp 86 – 95) Print ISBN:0-8186-711-43

[5] S. Ducasse, M. Rieger, and S. Demeyer , A language independent approach for detecting duplicated code, Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on 30 Aug 1999-03 Sep 1999(pp 109 – 118) ISSN :1063-6773,Print ISBN:0-7695-0016-1

[7] J. Mayland, C. Leblanc, and E. M. Merlo. "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", Proc. IEEE Int'l Conf. on Software Maintenance (ICSM) '96, pp. 244-253, Monterey, California, Nov. 1996

[8] Marcelo Sant Anna, Lorraine Bier. Clone Detection Using Abstract Syntax Trees, ICSM '98 Proceedings of the International Conference on Software Maintenance, page 368, 1998-03-16, IEEE Computer Society Washington, DC, USA ©1998, ISBN: 0-8186-8779-7

© 2015, IJARCSSE All Rights Reserved