Relation of Code Clones and Change Couplings

Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger

s.e.a.l. – software evolution and architecture lab,
Department of Informatics,
University of Zurich, Switzerland
{geiger, fluri, gall, pinzger}@ifi.unizh.ch

Abstract. Code clones have long been recognized as bad smells in software systems and are considered to cause maintenance problems during evolution. It is broadly assumed that the more clones two files share, the more often they have to be changed together. This relation between clones and change couplings has been postulated but neither demonstrated nor quantified yet. However, given such a relation it would simplify the identification of restructuring candidates and reduce change couplings. In this paper, we examine this relation and discuss if a correlation between code clones and change couplings can be verified. For that, we propose a framework to examine code clones and relate them to change couplings taken from release history analysis. We validated our framework with the open source project Mozilla and the results of the validation show that although the relation is statistically unverifiable it derives a reasonable amount of cases where the relation exists. Therefore, to discover clone candidates for restructuring we additionally propose a set of metrics and a visualization technique. This allows one to spot where a correlation between cloning and change coupling exists and, as a result, which files should be restructured to ease further evolution.

1 Introduction

Code duplication is often cited as one of the major *bad smells* in software systems [1]. Systems containing a large proportion of duplicated code are considered to be difficult to maintain. It is estimated that normal industrial source code contains 5 – 20 % of duplicated fragments [2]. The financial impact of maintenance is grave – the costs of changes carried out after delivery are estimated at 40 – 70 % of the total costs during a system's lifetime [3].

As bad smells are indicators for maintainability problems, they lose their significance if the system remains stable and is never changed after its initial release. According to Lehman's *Laws of Software Evolution* [4], software systems which are actively used to solve problems in the real world are never completely stable during their lifetime. Basically, a system has to evolve so that its users remain satisfied. In this case the possible negative influence of code clones on the maintainability comes into play. Code duplication increases the change effort and reduces the understandability of the code drastically. Thus, code clones are a major factor that have to be considered during the evolution of a system.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 411–425, 2006.

A sign of maintainability problems during the evolution of a system are change couplings. Gall *et al.* defined change couplings as files which are committed at the same time, by the same author, and with the same modification description [5]. If such couplings occur only sporadically, they do not present a major problem. If, on the other hand, files are frequently changed together during the evolution of the software system, a refactoring or even reengineering should be considered.

Based on the assumption that whenever a duplicated code fragment is changed its variants also have to change [6], there seems to be a strong relation between code clones and change couplings. In this paper, we investigate whether this relation holds. We present a framework to determine the relation of code clones and change couplings and introduce a visualization technique aiding developers to choose which code clones to refactor. We further present a validation of our framework with the large open source project Mozilla. The results of the validation show that although the relation is statistically unverifiable it derives a reasonable amount of results where the relation exists. Furthermore, it shows that based on the relation data our visualization technique can be used to identify the candidates for a refactoring.

The remainder of the paper is organized as follows: Section 2 presents related work that has been done in the area of code clone detection and the impact of these duplications to the evolution of software systems. In Section 3 we describe our framework that has been applied to the case study. Section 4 presents a validation of our framework and discusses its results. We conclude the paper in Section 5 and indicate areas for future research.

2 Related Work

A large number of code clone detection techniques have been developed. Four different general approaches can be discerned: detection based on lexical analysis [7, 8, 9], on source code metrics [2], on an abstract syntax tree representations of the system [10], or on isomorphic program dependence graphs [11]. Burd and Bailey give a comparison between the different approaches in [12]. Most of these approaches offer graphical user interfaces using dot plots to visualize the code clones. We worked with the Gemini environment for CCFinder [13] and with Duploc [14]. We found that the dot plot visualization technique was most useful for smaller fragments of source code but did not scale well for large systems such as Mozilla.

Casazza *et al.* describe the application of code clone detection tools on a large scale multi-platform software system in [15]. They explore the cloning percentages across different platform–dependent modules of the Linux kernel. The percentage of cloning that has been detected can be considered low. Compared to their approach we focus on the effect of code clones on change couplings.

Recent studies have shown why and how programmers introduce code clones into software systems [16] and how software development could benefit from the

inclusion of code clone detection tools into the development process [17]. The evolution of code clones has been investigated by Kim *et al.* in [18]. They provide a classification for evolving code clones but not for their impact on the change coupling behavior of the whole system. The main result of their paper is that code duplications cannot be considered inherently bad and do not need to be refactored in every case.

Work on the classification of code clones has recently been done by Kapser and Godfrey [19]. They propose a tool to interpret and classify the results gathered by code clone detection tools. Their case study also shows improvements in the elimination of false positive candidates returned by most clone detection tools.

The concept of the release history database (RHDB) was first described in [20] and [21]. The database uses version and bug tracking data and contains data obtained from the Mozilla open source project which uses CVS as version control system and Bugzilla for the organization of bug reports. Further work on logical and change couplings during the evolution of a software system was presented in [5, 22].

We adopted the visualization technique using polymetric views developed by Lanza and Ducasse [23] for the use with code clones and change couplings.

3 Framework

In this section we present our framework for analyzing the relation between code clones and change couplings. The framework consists of six steps as shown in Figure 1. The following subsections describe each step in detail.

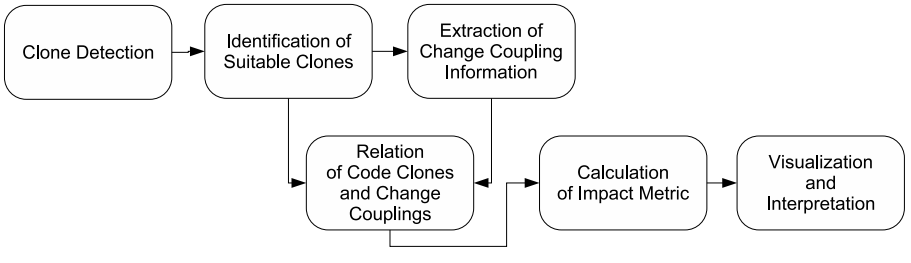

Fig. 1. Overview of the framework

3.1 Clone Detection

A pre-selection of three code clone detection tools has been made yielding the candidate tools Duploc [8], CloneDR [10], and CCFinder [9]. We selected the clone detection tool according to several criteria which we considered important for their applicability to the case study: language support for C and C++, maximal input size in lines of code, user interface, output format, recall, and precision. Most of these criteria are not directly measurable or even depend on the subjective perception of the user.

By means of these criteria we have chosen CCFinder as the clone detector most appropriate for our needs. Its recall, user interface, and output format fit best our needs to address our research goal. For an in-depth evaluation of the three clone detection tools we refer to [24].

3.2 Identification of Suitable Clone Candidates

As the goal of this framework is to define the relation between duplicated code and change couplings between files, the interesting clone pairs are those in which the cloned code fragments appear in two or more files. Furthermore, clones whose length varies or that appear or disappear during the examined period are considered more interesting than duplications that remain stable. This selection criterion is based on the assumption that there is a significant relation between code clones and change couplings.

Absolute numbers are inadequate when comparing different files because their lengths vary. The same applies to the length of cloned code fragments. Therefore our model of code clone classification relies on the clone coverage in every single file. This ratio is for file A compared to B defined as

$$CloneCoverage(A, B) = \frac{ClonedLines(A, B)}{NCLOC(A)}$$

where $ClonedLines(A, B)$ is the number of lines in file A that are clones of lines in file B. $NCLOC(A)$ is the number of lines of source code in file A not counting comments and blank lines. A cloned line is only counted once even if it is part of more than one clone pair or is covered multiple times by overlapping clones. When more than two files are compared, every pair of files out of this set has to be compared separately.

Two files A and B can share more than one semantically distinct clone pair. The types can be used to classify every instance of a clone pair. And in this paper, $CloneCoverage(X, Y)$ is always calculated for all code clones shared by X and Y.

To apply clone coverage to a set of evolving files, it is necessary to observe the clone coverage values over several versions of the files. These comparisons allow us the classification of each file pair sharing code clones into one of the following five types depending on the development of its clone coverage.

- **Type 0 (stable):** The relative length of cloned fragments in question remains the same between versions i and $i + 1$:

$$CloneCoverage(A, B)_i = CloneCoverage(A, B)_{i+1} \neq 0$$

- **Type 1 (new):** A clone is newly introduced in version $i + 1$:

$$CloneCoverage(A, B)_i = 0 \quad \wedge \quad CloneCoverage(A, B)_{i+1} > 0$$

- **Type 2 (removed):** A clone is removed between the versions i and $i + 1$:

$$CloneCoverage(A, B)_i > 0 \quad \wedge \quad CloneCoverage(A, B)_{i+1} = 0$$

– **Type 3 (increased):** Clone with increasing significance, *i.e.*, the clone coverage in version $i+1$ is larger than in version i:

$$CloneCoverage(A,B)_i < CloneCoverage(A,B)_{i+1}$$
$$\text{and} \quad CloneCoverage(A,B)_i > 0 \ \wedge \ CloneCoverage(A,B)_{i+1} > 0$$

– **Type 4 (decreased):** Clone with decreasing significance, *i.e.*, the clone coverage in version $i+1$ is smaller than in version i:

$$CloneCoverage(A,B)_i > CloneCoverage(A,B)_{i+1}$$
$$\text{and} \quad CloneCoverage(A,B)_i > 0 \ \wedge \ CloneCoverage(A,B)_{i+1} > 0$$

Types 1 to *4* indicate changes in code clones during evolution. Among them, those best suited for further investigation are clones of *Type 1* and *2*. We expect that the change couplings between files containing cloned fragments of each other show a relation between the changing code clones and their later couplings. If this assumption holds, for example two files into which a *Type 1* clone is introduced after version i are expected to exhibit more change couplings in subsequent versions. *Type 0* clones are also of interest because according to the hypothesis, change couplings caused by code clones are expected to be stable.

3.3 Extraction of Change Coupling Information

For this step of the framework we relied on our previous work on the release history database (RHDB) described in [20]. The RHDB contains data obtained, for example, from the Mozilla open source project. In particular, it stores data about the modification reports obtained from versioning control systems (CVS) repository of Mozilla and problem report data obtained from Mozilla's Bugzilla database.[1] In our framework we can exploit the RHDB to retrieve the change coupling data for the files that share code clones.

The number of change couplings between a pair of files (or similar entities of source code) during a given interval is the same for each file of a change coupled pair. The number of check–ins during the same time interval can, however, vary giving us a distinct ratio for each file. The coupling coverage metric we subsequently use is defined as

$$CouplingCoverage(A,B,I) = \frac{ChangeCouplings(A,B,I)}{Checkins(A,I)}$$

where $ChangeCouplings(A,B,I)$ is the number of times files A and file B are checked in together during time interval I and $Checkins(A,I)$ is the total number of times file A is checked in during I.

[1] http://bugzilla.mozilla.org

3.4 Relation of Code Clones with Change Couplings

A potential relation between code clones and change couplings is based on the assumption that pairs of source files sharing code clones are changed together [6]. This assumption has been taken for granted but not yet been proven.

For the investigation of this assumption we use the code clone and change coupling coverage values of each file pair and relate them. Results are represented in a dot plot where each dot refers to a file pair sharing code clones. The position of a dot is computed by drawing the code clone coverage value on the X-axis and the change coupling coverage value on the Y-axis.

Based on the assumption stated before we expect a concentration of dots along the diagonal meaning that low clone coverage leads to few change couplings and high clone coverage leads to frequent change couplings. An example of such a dot plot is depicted by Figure 3 in Section 4. And, as will be shown in the case study, the expectation is not always fulfilled.

To enable an interpretation of resulting dot plots we use regression analysis to quantify the relation between code clones and change couplings. In this paper we consider linear and logarithmic regression analysis. Two premises must be fulfilled for the regression to be significant. One is that a representative sample of files containing code clones is available for the calculation. The second is that this sample can be described with sufficient precision by a regression function, meaning that the correlation coefficient is close to 1.

3.5 Definition of a Metric to Describe the Impact of Code Clones

The relation presented before is based on the relative values of code clone and change coupling coverage. In addition the absolute length of a clone as well as the total number of change couplings influence our impact metric. That means, a longer fragment of duplicated code tends to have a larger influence than a shorter sequence. Furthermore, a file that is changed more often has a bigger potential of presenting a problem than a file that is never touched during the evolution of a system.

Based on these assumptions we select the following input parameters for the calculation of our impact metric:

- Clone coverage,
- Coupling coverage,
- Length of cloned fragments, and
- Absolute number of coupled check–ins.

Because of the difficulty to express the four parameters in one view a light–weight approach is used applying Lanza's polymetric views [23]. The key idea of polymetric views is to map metric values to graphical attributes, such as shape, size, and color of glyphs to activate the visual recognition capabilities of humans.

In our visualization, the four metrics listed above can be displayed in a Cartesian coordinate system enriched with additional use of color and the diameter of circles in the chart. The mapping of metric values to graphical attributes is depicted Figure 2.

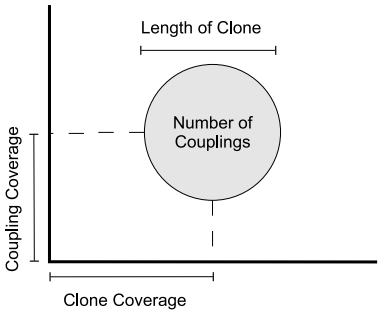

Fig. 2. Description of the metrics used in the visualization

The size of a circle is defined in proportion to the length of the clones. The maximum diameter is fixed and corresponds to the length of the longest clone. All other diameters are calculated proportionally to the length of the rest of the clones:

$$Diameter(A) = MaxDiameter \cdot \frac{ClonedLines(A,B)}{max(ClonedLines(X,Y))}$$

where $MaxDiameter$ is a constant describing the maximal diameter of a circle and $max(ClonedLines(X,Y))$ is the maximum length of cloned fragments to be visualized.

The fill color of a circle is defined in a way that the highest number of couplings is displayed as red. The intermediate colors are determined by variations of the RGB value proportional to the relative number of couplings so that a gradual transition to blue is achieved, which corresponds to zero couplings. The R and B–values are calculated by

$$R = \frac{ChangeCouplings(C,D,I)}{max(ChangeCouplings(X,Y,I))} \cdot 255, \quad \text{and } B = 255 - R$$

where R is the RGB–value for red and B the RGB–value for blue of the color of the circle in the chart. C and D are the specific files under consideration. $max(ChangeCouplings(X,Y,I))$ represents the maximal number of change couplings between any two files X and Y during interval I.

Unlike a numerical approach, this visualization is not dependent on a significant regression. The user is able to see possible problems and to react by closer inspection of the affected files.

4 Framework Validation

For the validation of our framework we applied the tools and methods to the open source project Mozilla[2]. The following sections report on our experiences and present the results of the experiments and the insights gained.

[2] http://www.mozilla.org/

4.1 Clone Detection

For the detection of code clones we selected the CCFinder tool. In our framework these output files form the basic input to calculate the correlations between code clones and change couplings.

The input data for our clone detection comprised the seven source code releases of Mozilla: 0.92, 0.97, 1.0, 1.3a, 1.4, 1.6, and 1.7. The release period between these releases is about 6 months. For each release we selected the ".c" and ".cpp" files that contain most parts of the implementation. We also skipped the header (".h") files because these files mostly contain only declarations and no implementation of functionality. The following description of the case study is based on the input data of Release 1.7 comprising 5,882 files with 2,980,000 lines of code (LOC).

For the configuration of the CCFinder tool we performed a number of test runs and came up with two possible configurations for the minimal length of code clones which are 30 and 70 tokens. 70 tokens were used when processing large amounts of data, such as all files of one release, to allow us to visualize the code clones. Otherwise, for our analysis of the relation between code clones and change coupling we used 30 tokens. Using 30 tokens results in code clones with a minimal length of 2.9–3.9 lines of C or C++ source code. Figure 3 shows the dot plot of detected code clones in source files of Mozilla Release 1.7 generated with the Gemini tool [13].

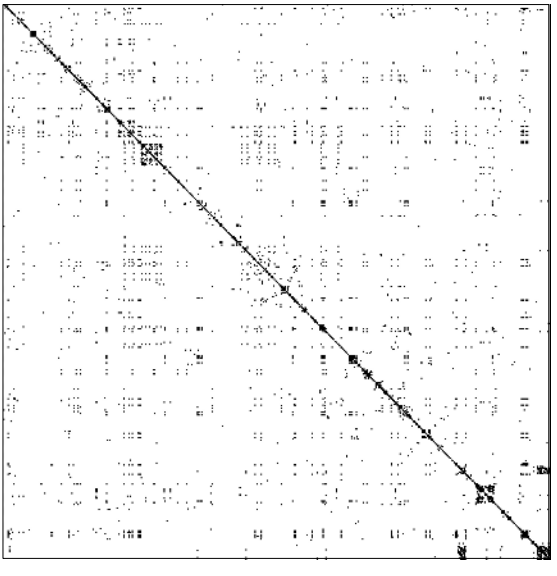

Fig. 3. Dot plot of code clones in Mozilla Release 1.7 (70 tokens)

In total the CCFinder tool detected 661,861 code clones in the source files of Release 1.7. In the dot plot files are arranged on a directory-basis allowing us to identify inter- and intra-module clones. Code clones within modules are indicated by the clusters positioned around the diagonal line. For instance, the cluster in the lower right corner shows the code clones within the "GFX and Widget–Mac" module. The other clusters in the dot plot indicate inter-module code clones.

4.2 Identification of Suitable Clone Candidates

Not all code clone candidates that are detected by CCFinder can be used for the purpose of this case study. One problem are false positives or clones only consisting of sequences of `#include`–statements, declarations of variables, or `switch`–statements. For the relation between code clones and change couplings *Types 1* to *4* are of interest because they changed during the evolution.

We selected a representative sample of 31 files to examine the types of clone containing file pairs occurring in the case study. These files form 21 file pairs of which almost none are of a single type of file pair within the examined interval. *Type 0* file pairs occur in 13, *Type 1* and *2* in 2, *Type 3* in 11, and *Type 4* in 12 pairs. Noteworthy, example pairs for *Type 0* and *4* file pairs are {`nsMathMLmoverFrame.cpp`, `nsMathMLmunderoverFrame.cpp`}, and {`os2/nsFilePicker.cpp`, `windows/nsFilePicker.cpp`} respectively.

Since in this case study most of the clone pairs occur in various types of file pairs, we did not consider the selection of special clone candidates. Therefore, we input all detected clones to the relation analysis.

4.3 Extraction of Change Coupling Information

For the extraction of the change coupling information we considered all files of Mozilla Release 1.7 that share at least one code clone. With this set of files we accessed our release history database (RHDB) and retrieved the change coupling information.

There is one major difference between the examination of code clones and that of change couplings that we have to consider: code clones involve the exploration of files at a given point in time – the date of each release of Mozilla – while the latter must be investigated over a certain time interval (*i.e.*, between two or more Mozilla releases).

Summarized we retrieved 139,523 change coupling records from the RHDB for all seven Mozilla releases (up to Mozilla 1.7).

4.4 Relation of Clone Data and Change Couplings

For relating the detected code clones with extracted change couplings we computed per file pair the code clone coverage and the change coupling coverage (see Section 3). The two coverage values then were plotted against each other yielding to the dot plot shown in Figure 4.

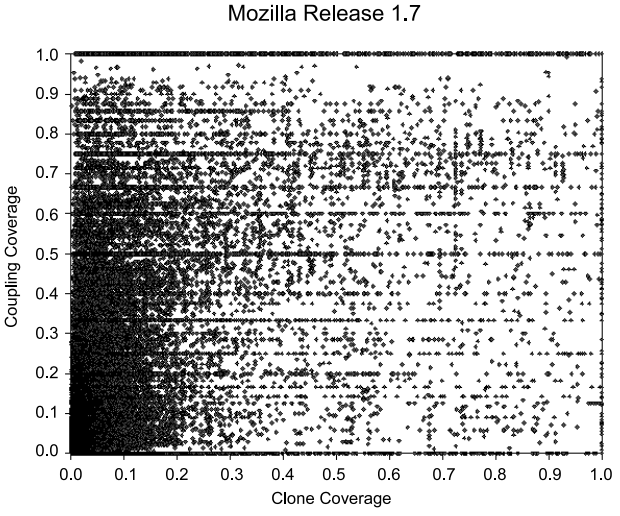

Fig. 4. Relation between code clones and change couplings in Mozilla 1.7

The concentration of values where the clone coverage ratio is below 0.2 indicates that the relation between clone and coupling coverage is pretty much random and difficult to interpret.

To prove the relation between code clones and change couplings we applied two types of (straightforward) regression analysis: linear and logarithmic regression analysis; using other functions are subject of future work. Because of the huge amount of data we started the regression analysis with three random samples of 65,536 file pairs sharing code clones. In each case, the R^2 value was better for linear than for logarithmic regression over the same sample. A linear regression resulted in the best fitting function with the highest coefficient of determination of 0.702:

$$CouplingCoverage(A, B, I) = 1.038 \cdot CloneCoverage(A, B) + 0.097$$

The resulting equation explains 70.2 % of the scattering visible in the chart. The other attempts at regression analysis yielded lower R^2 values.

Similar regression analyses were computed for the 30,433 instances of data where the clone coverage exceeded the threshold of 0.2. A linear regression with a low R^2 value of 0.2088 resulted in the equation

$$CouplingCoverage(A, B, I) = 0.512 \cdot CloneCoverage(A, B) + 0.4781$$

which is not a close fit compared to the results obtained by a sample of all input values.

The findings of our analysis indicate a certain relation between cloned fragments of source code and change couplings during evolution of the software. This connection was expected from previous work starting with [1]. Usually the larger

the clone coverage between two files is, the more often these files are coupled. However, based on our regression analyses it is neither possible to conclude that code duplications are reflected in high change coupling coverage values nor is the opposite true. From the results of this case study it is impossible to definitely exclude the possibility that there is in fact no statistically relevant correlation between code duplications and change couplings.

Change couplings can have causes other than code clones. Files fulfilling similar roles in the system often are changed together even though they might not contain many duplicated code fragments. Despite these exceptions, the general tendency for files with a high clone coverage value is to be coupled more often than files with a lower percentage of duplications.

An examination of clone and coupling coverage can be used to identify groups of files that would benefit from a determined refactoring effort. In Mozilla, we identified several such candidates. An example is the file `nsMathMLmsubFrame.cpp` of the MathML module which is coupled with files `nsMathMLmsubsupFrame.cpp` and `nsMathMLmsupFrame.cpp` in the same folder every single time it is changed between Releases 0.92 and 1.7.

Using our relation analysis it is not possible to distinguish harmless from dangerous code duplications simply by looking at the results of a code clone detection run on only one release of a software system. It is, however, safe to say that the larger the clone coverage is, the higher is the probability of it becoming "dangerous" during evolution. To express this degree of "danger" we applied our visualization technique presented in Section 3.

4.5 Visualizing the Impact of Code Clones

Because of the difficulties of establishing a sound mathematical correlation between code duplications and change couplings we applied the polymetric views visualization technique described in Section 3. This provided us with more insights into the relation and in particular pointed out file pairs with a strong relation being the candidates for a refactoring.

Figure 5 and Figure 6 depict the polymetric views created for the two modules MathML and JPEG of Mozilla Releases 0.9.2 and 1.7.

The MathML module consists of 26 C++ files between which 470 distinct pairs of files share duplicated code. Figure 5 depicts the situation for MathML. Both, the left and the right chart point out the 5 files E, F, G, H, and I in the upper right corner. They are frequently coupled with other files and share large fragments of duplicated code as indicated by the size of the circles. Files containing relatively few clones and with low code clone and change coupling coverage are drawn on the left side of the chart. By comparing the charts of both releases we distinguish the different types of (clone containing) file pairs (see Section 3.2).

For instance, the total length of clones as well as the clone coverage in L significantly decreased from Release 0.9.2 to 1.7 indicating a reengineering effort. According to our classification this is a good example for a *Type 4* file pair. Further *Type 4* file pairs are B, E, J, and K. In contrast, F represents a good

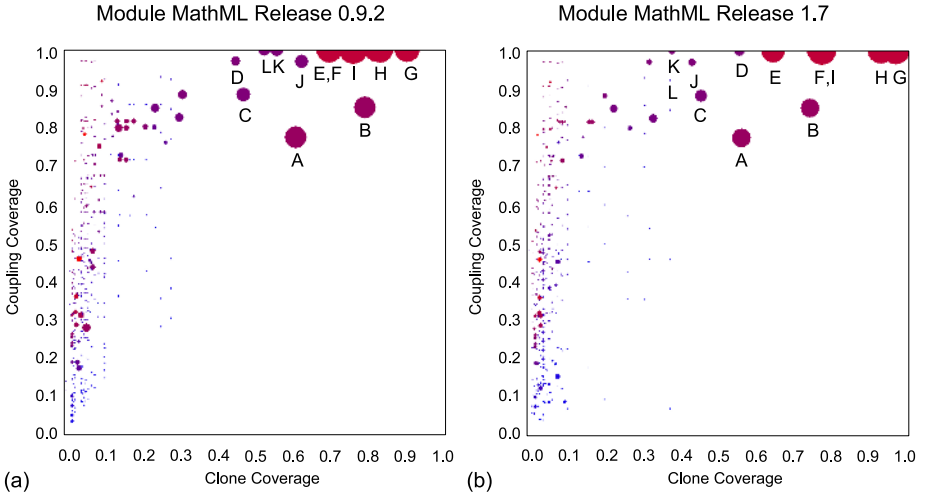

Fig. 5. Visualization of Mozilla module MathML of Releases 0.9.2 (a) and 1.7 (b)

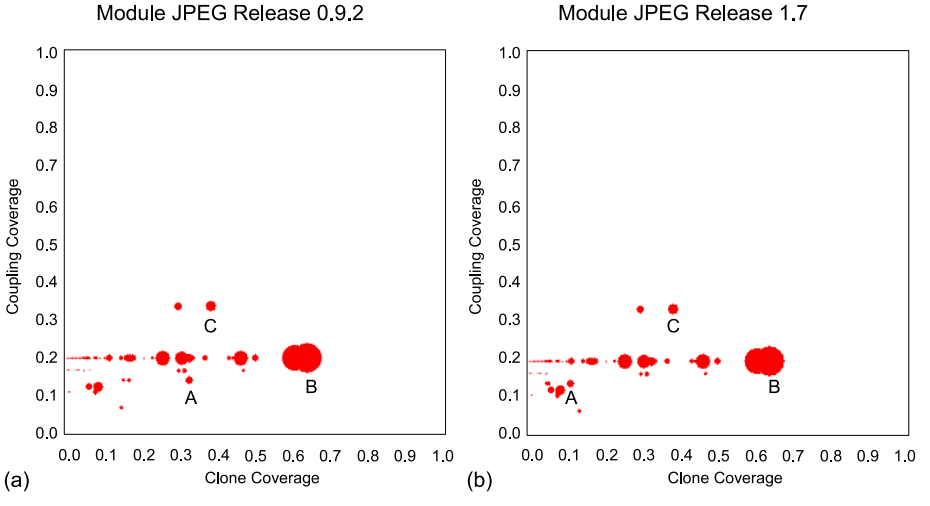

Fig. 6. Visualization of Mozilla module JPEG of Releases 0.9.2 (a) and 1.7 (b)

example of a *Type 3* file pair as indicated by an increasing clone coverage value. A similar trend can be seen for the file pairs represented by A, C, D, G, H, and I.

The situation for Mozilla's JPEG module is different as depicted by Figure 6. The 52 C files of this module form 230 distinct file pairs sharing code clones. In both charts most of the circles are equally red (dark) because every file of the module was coupled exactly once during the observed time periods. In this case,

the selection of candidates for a refactoring relies on the length of code clones and the clone coverage alone. The glyphs in both graphs show a large number of *Type 0* file pairs, for example B and C. Furthermore, there are few other types of file pairs, such as A, showing a relatively stable module JPEG.

Summarized, based on the metric values of file pairs sharing clones our polymetric views allowed us to spot the degree of "danger" of code clones. The most "dangerous" code clones were highlighted pointing us to the candidates in which code clones resulted in high change couplings, *e.g.*, H and G in Figure 5. They are first-class candidates to refactor.

5 Conclusions and Future Work

It is broadly assumed that the more clones two files share, the more often they have to be changed together. We address this problem of qualifying change couplings via code clone analysis.

In this paper, we discussed the relation of code clones and change couplings taken from release history data to examine whether a correlation exists between the two. For that, we proposed a framework to examine code clones and relate them to change couplings. The individual steps include clone detection and classification of clones into clone types, extraction of change couplings for the files in which the clones exist, calculating the relation between clones and change couplings, and computing and visualizing a relation metric to identify restructuring candidates.

We validated our framework with the open source project Mozilla and the results of the validation show that although the relation is statistically indeterminable it derives a reasonable number of cases where such a relation exists.

Our framework is not limited to the Mozilla case study; it is essentially independent of the type of system or of the programming language in which the system is written. The metrics defined are relatively simple yet effective to compute and require access to the system's source code and to a release history database containing release, modification, and bug report data.

We use polymetric views as a visualization technique to detect problematic code clones. This allows one to spot where a correlation between cloning and change coupling exists and, as a result, which files should be restructured to ease further evolution. If such a framework is integrated into a software engineering environment, it could potentially offer a useful guidance in the decision which clones are to be refactored. This is subject of our current work.

A result of this paper is that at least in the Mozilla case study, the correlation between code clones and change couplings is too complex to be expressed easily. For a significant distinction between clones that are irrelevant to the evolution of a system and clones that are harmful, more information is needed than what can be obtained automatically. Despite sophisticated tools that are available, the judgement of the software engineer is still needed.

As future work we plan to further improve the examination and visualization of the relation between code duplications and change couplings to distill all those

parts of a system in which clones are the cause for change couplings. We will further integrate this kind of analysis with our other evolution analysis tools to enable a more comprehensive picture by combining change dependencies, bugs, and code clones.

References

1. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
2. Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In: Proceedings of the International Conference on Software Maintenance (ICSM), Monterey, CA, USA, IEEE CS (1996) 244–253
3. Grubb, P., Takang, A.A.: Software Maintenance – Concepts and Practice. 2nd edn. World Scientific (2003)
4. Lehman, M.M., Belady, L.: Program Evolution Processes of Software Change. Academic Press (1985)
5. Gall, H., Hajek, K., Jazayeri, M.: Detection of Logical Coupling based on Product Release History. In: Proceedings of the 14th International Conference on Software Maintenance (ICSM), Bethesda, Maryland, USA, IEEE CS (1998) 190–198
6. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann, San Francisco, CA, USA (2003)
7. Baker, B.S.: A Program for Identifying Duplicated Code. Computing Science and Statistics **24** (1992) 49–57
8. Ducasse, S., Rieger, M., Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code. In: Proceedings of the 15th International Conference on Software Maintenance (ICSM), Oxford, England, UK, IEEE CS (1999) 109–118
9. Kamiya, T., Kusumoto, S., Inoue, K.: CCfinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. IEEE Transaction on Software Engineering **28** (2002) 654–670
10. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In: Proceedings of the 14th International Conference on Software Maintenance (ICSM), Bethesda, Maryland, USA, IEEE CS (1998) 368–377
11. Krinke, J.: Identifying Similar Code with Program Dependence Graphs. In: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE), Stuttgart, Germany, IEEE CS (2001) 301–310
12. Burd, E., Bailey, J.: Evaluating Clone Detection Tools for Use during Preventative Maintenance. In: Proceedings of the 2nd International Workshop on Source Code Analysis and Manipulation (SCAM), Montreal, Canada, IEEE CS (2002) 36–43
13. Ueda, Y., Kamiya, T., Kusumoto, S., Inoue, K.: Gemini: Maintenance Support Environment Based on Code Clone Analysis. In: Proceedings of the 8th International Symposium on Software Metrics (METRICS), Ottawa, Canada, IEEE CS (2002) 67–76
14. Rieger, M., Ducasse, S.: Visual Detection of Duplicated Code. In: Workshop on Object-Oriented Technology, Brussels, Belgium, Springer-Verlag (1998) 75–76
15. Casazza, G., Antoniol, G., Villano, U., Merlo, E., Penta, M.D.: Identifying Clones in the Linux Kernel. In: Proceedings of the 1sr International Workshop on Source Code Analysis and Manipulation (SCAM), Florence, Italy, IEEE CS (2001) 90–97

16. Kim, M., Bergman, L., Lau, T., Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In: Proceedings of the International Symposium on Empirical Software Engineering (ISESE), Redondo Beach, CA, USA, IEEE CS (2004) 83–92
17. Lague, B., Proulx, D., Mayrand, J., Merlo, E.M., Hudepohl, J.: Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In: Proceedings of the 13th International Conference on Software Maintenance (ICSM), Bari, Italy, IEEE CS (1997) 314–323
18. Kim, M., Sazawal, V., Notkin, D.: An Empirical Study of Code Clone Genealogies. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE), Lisbon, Protugal, ACM Press (2005) 187–196
19. Kapser, C., Godfrey, M.W.: Aiding Comprehension of Cloning Through Categorization. In: Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE), Kyoto, Japan, IEEE CS (2004) 85–94
20. Fischer, M., Pinzger, M., Gall, H.: Populating a Release History Database from Version Control and Bug Tracking Systems. In: Proceedings of the 19th International Conference on Software Maintenance (ICSM), Amsterdam, The Netherlands, IEEE, IEEE CS (2003) 23–32
21. Gall, H., Jazayeri, M., Krajewski, J.: CVS Release History Data for Detecting Logical Couplings. In: Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE), Helsinki, Finland, IEEE CS (2003) 13
22. Fluri, B., Gall, H.C., Pinzger, M.: Fine-Grained Analysis of Change Couplings. In: Proceedings of the 5th International Workshop on Source Code Analysis and Manipulation, Budapest, Hungary, IEEE CS (2005) 66–74
23. Lanza, M., Ducasse, S.: Polymetric Views – A Lightweight Visual Approach to Reverse Engineering. IEEE Transactions on Software Engineering **29** (2003) 782–795
24. Geiger, R.: Evolution Impact of Code Clones – Identification of Structural and Change Smells based on Code Clones. Master's thesis, University of Zurich (2005) http://seal.ifi.unizh.ch/da.