Analysis and Visualization for Clone Refactoring

Minhaz F. Zibran
Department of Computer Science, University of New Orleans, LA, USA
Email: zibran@cs.uno.edu

***Abstract*—Clone analysis and visualization help in understanding characteristics of clones and indicate potential clones as cost-effective candidates for refactoring. Many studies have analyzed clones and their evolution while a number of techniques have also been proposed for visualizing clones for aiding clone analysis. However, clone analyses and visualizations with respect to inheritance hierarchy and call graphs have remained ignored so far. In this position paper, we argue that such analyses and visualizations are necessary to help in dealing with clones for refactoring.**

I. Introduction

Duplicated code, or code clone is a well-known code smell. Software systems typically have 9%-17% [21] duplicated code, up to even 50% [14]. Due to the negative impacts (e.g., bug creation, bug propagation, code instability, code inflation) of code clones, it often becomes necessary to remove them by careful refactoring.

Till date, clone refactoring is still highly dependent on human efforts, which is error-prone and can be inefficient as well. Effective refactoring of code clones requires a proper understanding of the distributions and dependencies among the program components where the clones reside in. Support for performing clone analyses and visualization can help the developers in choosing and performing appropriate refactorings.

Until recently, many studies have explored the evolution, distribution, and characteristics of code clones, which contributed to the understanding of code clones, reasons for code cloning, the nature of cloned code and their implications in the development and maintenance of a software system. However, the analyses of the distribution of code clones remained limited within the dispersion of clones with respect to the file-system hierarchy only. In this position paper, we argue that from the perspective of clone refactoring, the support for analysis and visualization of clones with respect to the inheritance hierarchy and call graphs is a necessity, which is largely ignored so far.

II. State of the Art

Visualization support helps in clone analysis by facilitating quick view over the characteristics of clones. Thus, many clone visualization techniques have been proposed in the literature as summarized in Table I. As shown in Table I, the existing clone presentation techniques can be viewed to have two criteria [8]. The first criterion (middle column in Table I) refers to the level of granularities (such as at the code segment level or file level or subsystem level) at which the clone entities are visualized.

TABLE I
Summary of clone visualization techniques

Visualization Techniques	Clone Granularity	Clone Relation
Tree Map [1], [14]	F, S	G
Scatter Plot [1], [14], [17]	F, S, C	P
System Model View [14]	F, S	P
Clone System Hierarchical Tree [7]	F, S	P, G
Hasse Diagram [9]	F	G
Clone Group Family Enumeration [14]	F	G
Duplication Web [14]	F	P
Dependency Graph [11]	S	P
Hierarchical Dependency Graph [1]	S	P
Clone Coupling and Cohesion [8]	S	S_c
Metric Graph [17]	C	G
Clone Cluster View [4]	C	G
Hyper-Linked Web Page [2], [10]	C	G
Clone Visualizer View [16]	F, C	G
Stacked Bar Chart [19]	F, C	G
Line Chart [19]	F, C	G
Edge Bundle View [5]	F, S	P

Here, F = file, S = sub-system or sub-directory, C = code segment
P = clone-pair, G = clone-group, S_c = super-clone

The second criterion (the rightmost column in Table I) refers to the type of clone relationship addressed by the presentation, that is, whether clones are showed at the clone-pair level or grouped into clone-sets or super clones. A *super clone* is an aggregated representation of multiple clone-groups residing in the same source code entity (e.g., file).

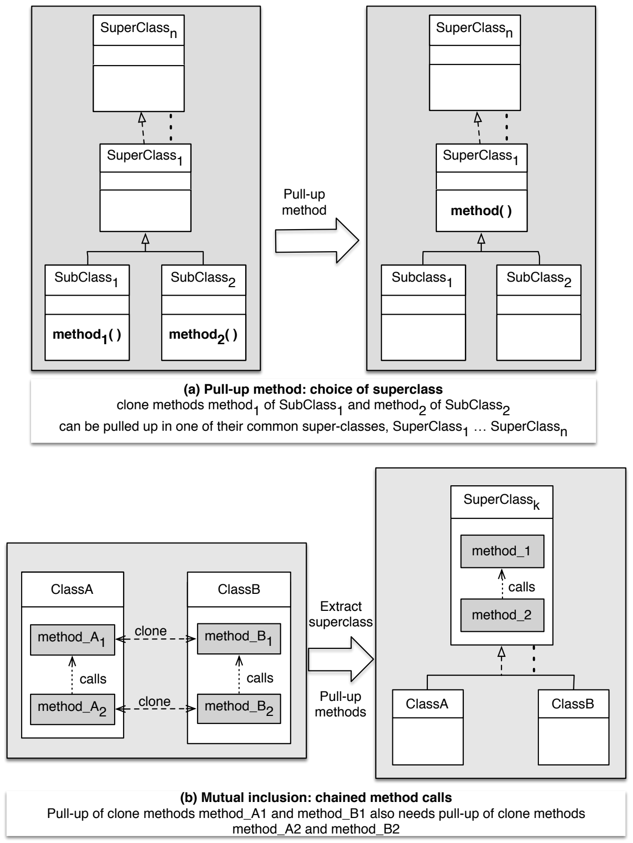
Similar to the *analyses* of dispersion of clones, the *visualization* of clones' distributions also remained confined with respect to the file-system hierarchy, as can be inferred from Table I. Although existing clone analyses and visualizations have considered different granularities and clone relationships, the analysis and visualization of the distributions of clones with respect to inheritance hierarchy and call delegations have remained ignored.

III. Analysis and Visualization for Refactoring

Among many general refactoring patterns [3], *Extract Method*, *Pull-up Method*, and *Extract Class* are the most prominent patterns that can be applied for clone refactoring [12], [13], [20], while in the literature, some variations (e.g., *Extract Superclass, Form Template Method, Extract Utility Class, Move Method*) of these refactoring patterns are also found to have been considered for the same.

Note that the clone refactoring patterns are formulated on the basis of program structure in terms of inheritance

Accepted for publication by IEEE. © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.



(a) Pull-up method: choice of superclass
clone methods method$_1$ of SubClass$_1$ and method$_2$ of SubClass$_2$
can be pulled up in one of their common super-classes, SuperClass$_1$ … SuperClass$_n$

(b) Mutual inclusion: chained method calls
Pull-up of clone methods method_A1 and method_B1 also needs pull-up of clone methods method_A2 and method_B2

Fig. 1. Examples of Clone Refactoring

hierarchy and method call delegations. For determining the appropriate refactoring strategies, the developer must invest a significant effort in understanding the context, constraints (e.g., dependencies), and locations of the clones within the inheritance hierarchy and method call graphs [20].

Figure 1 presents simple examples of clone refactoring within the context of inheritance hierarchy and method call graph. The example in Figure 1(a) shows that the developer must choose the appropriate superclass in the inheritance hierarchy to apply *Pull-up Method* refactoring. The example in Figure 1(b) demonstrates a context where the developer needs to identify and deal with the dependencies among the clones (mutual inclusion [20] in this case, aka, chained clones [18]). For demonstrating the ideas, the figure presents simple instances while more complicated scenarios appear in the source code of real-world software systems. Thus, support for interactive visualization and analysis of such scenarios of clones' dispersion and dependencies can offer immense help in clone refactoring. In addition, such visualizations can also help in quick exposure of program design flaws such as lack of cohesion, improper object-orientation, and issues with modularity.

IV. Conclusion

Existing visualization techniques can be fortified for clone visualization in the inheritance hierarchy and call graphs. For example, considering inheritance as a subset relationship, interactive *TreeMap* [1], [14] can be used for visualizing clones in inheritance hierarchy instead of traditional *TreeMap* view of clones in the file-system hierarchy. Customization of *hierarararchical dependency graph* [1] or simply *dependency graph* [11] can also be suitable alternatives for visualizing clones in call graphs or inheritance hierarchy.

As much as 80% of software costs are spent on maintenance [6]. During a maintenance task, most of the developer's effort is invested in understanding the underlying program structure and source code, while as high as 62% of such effort is typically wasted in investigating irrelevant parts (e.g., source files) of the program [15]. With appropriate metaphors, actionable support for clone analysis and visualization with respect to the inheritance hierarchy and call graphs can help in making better design decisions during clone refactoring and thus can minimize clone refactoring cost, which in turn can reduce the software maintenance cost as a whole.

References

[1] M. Asaduzzaman, C. Roy, and K. Schneider. VisCad: flexible code clone analysis support for NiCad. In *IWSC*, pages 77–78, 2011.
[2] J. Cordy, T. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *CASCON*, pages 1–12, 2004.
[3] M. Fowler, K. Beck, J.Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.
[4] Y. Fukushima, R. Kula, S. Kawaguchi, K. Fushida, M. Nagura, and H. Iida. Code clone graph metrics for detecting diffused code clones. In *APSEC*, pages 373–380, 2009.
[5] B. Hauptmann, V. Bauer, and M. Junker. Using edge bundle views for clone visualization. In *IWSC*, pages 86–87, 2012.
[6] Research Triangle Institute. The economic impacts of inadequate infrastructure of software testing. RTI Project Report 7007.011, National Institute of Standards and Technology, 2002.
[7] Z. Jiang and A. Hassan. A framework for studying clones in large software systems. In *SCAM*, pages 203–212, 2007.
[8] Z. Jiang, A. Hassan, and R. Holt. Visualizing clone cohesion and coupling. In *APSEC*, pages 467–476, 2006.
[9] J. Johnson. Visualizing textual redundancy in legacy source. In *CASCON*, pages 32–41, 1994.
[10] J. Johnson. Navigating the textual redundancy web in legacy source. In *CASCON*, pages 16–25, 1996.
[11] C. Kapser and M. Godfrey. Improved tool support for the investigation of duplication in software. In *ICSM*, pages 305–314, 2005.
[12] S. Lee, G. Bae, H. Chae, D. Bae, and Y. Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Software - Practice and Experience*, 41(5):521–550, 2010.
[13] H. Liu, G. Li, Z. Ma, , and W. Shao. Conflict-aware schedule of software refactorings. *IET Software*, 2(5):446–460, 2008.
[14] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109, 2004.
[15] Z. Soh, F. Khomh, Y. Gueheneuc, and G. Antoniol. Towards understanding how developers spend their effort during maintenance activities. In *WCRE*, pages 152–161, 2013.
[16] R. Tairas, J. Gray, and I. Baxter. Visualizing clone detection results. In *ASE*, pages 549–550, 2007.
[17] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *METRICS*, pages 67–76, 2002.
[18] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On refactoring support based on code clone dependency relation. In *METRICS*, pages 16–25, 2005.
[19] Y. Zhang, H. Basit, S. Jarzabek, D Anh, and M. Low. Query-based filtering and graphical view generation for clone analysis. In *ICSM*, pages 376–385, 2008.
[20] M. Zibran and C. Roy. Conflict-aware optimal scheduling of code clone refactoring. *IET Software*, 7(3):167–186, 2013.
[21] M. Zibran, R. Saha, M. Asaduzzaman, and C. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *ICECCS*, pages 295–304, 2011.