A Taxonomy of High Level Software Clones

Niyati Baliyan
School of ICT
Gautam Buddha University
Greater Noida, India
niyati.baliyan@gmail.com

Dr. Vidushi Sharma
School of ICT
Gautam Buddha University
Greater Noida, India
vidushi@gbu.ac.in

Abstract— **The idea of software clones is linked with similarity, which can occur at various levels of abstraction. Simple clones i.e. clones at lower levels of abstraction are classified in literature, however; high level clones are not. We propose and exemplify a consolidated yet novel taxonomy of high level clones. Our discussion augments thus far knowledge of high level clone types and their specialized detection techniques. Clone taxonomies can help in further analysis of high level clone phenomenon like in prioritizing clones for reengineering and for specifying reengineering guidelines.**

Keywords-High level similarities, software maintenance, classification, code clones.

I. INTRODUCTION

Since cloning in software systems is known to deter the software maintenance process [1], several techniques have been proposed to detect similar code fragments namely *simple clones* [2]; however, analysis of similarities at higher levels of abstraction still remains a nascent area. Detection of design-level similarities in software could further ease maintenance, and also aid in identification of reuse opportunities [3]. Clone taxonomies can help in prioritizing clones for reengineering and for specifying reengineering guidelines.

Most literature on clones is saturated with research on various aspects of simple clones or code clones i.e. similar program fragments. However, there may be similarities in certain higher levels of abstraction in software. If one applies a traditional waterfall model for software development; higher abstraction levels map to analysis and design phase, while lower abstraction levels map to coding, testing and maintenance phase. Therefore, by higher level similarity in software, it is meant that the similarity is at behavior, concept or domain model level, or simple clones occur at close proximity [4].

High level code clones convey important information about a software system design and implementation, their analysis can provide useful insights into program understanding, evolution, reuse and reengineering [5]. Classifying high level clones is an important first step towards further analysis of this phenomenon.

In this paper, we make the following contributions:

- Based on certain examples, we present possible categorization of high level software clones into-behavior, concept, UML domain model and collocated simple clones.
- We give restrictive definitions for subtypes of behavior clones and name them order 1, order 2 and order 3.

Having understood the significance of analyzing high level similarities to program understanding, evolution, reuse and reengineering, it is imperative to give unambiguous definitions of high level clone types [6]. Further, these clone classes shall direct specific clone detection and removal efforts, the ultimate aim of which is a better maintained software.

II. CLASSIFYING HIGH-LEVEL CLONES

Low (source code) level similarities - simple clones have been classified in literature as:

Type 1: This is an exact copy with no transformations except reformatting [7].

Type 2: This is a syntactically identical copy, with replacements of identifiers and literals.

Type 3 clones: This is a copy with further modifications (in statements/expressions)

Clones beyond Type 3: This is a copy with additional semantic preserving transformations.

Transliterated clones: These are programs transliterated in another programming language.

However, similarities at high levels of abstraction in software have not thus far been classified. We identify four major classes of such clones, as shown in Figure 1.

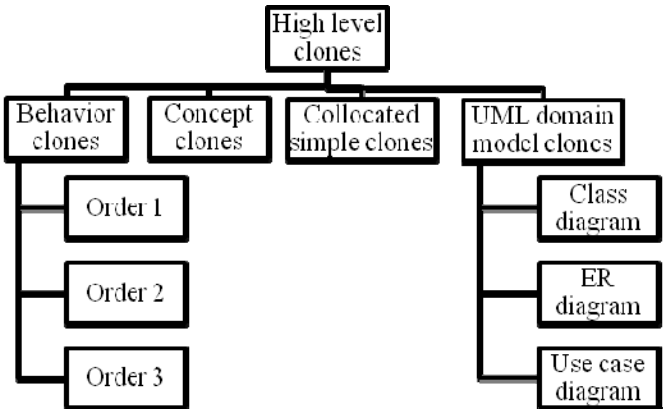

Figure 1. Proposed taxonomy of high-level clones

We have identified these classes of clones on the basis of available literature and we present our own examples of each of these classes. There may be many other possible high level similarities, and also instances of some of these classes may overlap with each other in real life scenarios. However, we aim to define the classification in a mutually exclusive and exhaustive manner. Various subclasses of behavior (semantic) clones and UML domain model clones are also discussed in Section 2.1 and 2.4 respectively.

Broadly speaking, behavior clones are functionally similar codes i.e. they perform the same function. Further, we define order 1, 2 and 3 of semantic clones. In section 2.2, we describe concept clones which originate from similar high level concepts like design pattern or mental template or even analysis model. Basit et al [8] define recurring patterns of simple clones to be structural clones. On the same lines, we introduce the term collocated simple clones to indicate simple clones occurring at various granularities like method, file and directory, thus we may abstract them as high level similarities. Section 2.3 describes UML domain model clones, which are similarities in domain models of two or more software systems. We may map types of UML domain models to subclass of this class of high level clones. Thus, we obtain class diagram clones, ER diagram clones and use case diagram clones.

Analyzing high level clones has an important role to play in program understanding, evolution, reuse and reengineering, and the first step in aiding this analysis is their classification and clear definition. We provide an insight into important yet nascent area of high level clone analysis. We discuss each class of high level clone in detail below.

2.1. Behavior Clones

Consider the problem of swapping two variable values, Figure 2 shows two classes which differ in representation but solve the same problem. The encircled code in Swap_no_temp swaps a, b without the use of a temporary variable, while the encircled code in Swap_temp does so, using a temporary variable temp. We verify that the end result of executing both codes is same.We looked at the above example in terms of input output behavior similarity. In simple terms, codes that give similar output on being fed similar inputs are called behavior clones. They may or may not be representationally similar, but often their independent development often creates redundancy [9]. For practical purposes, we consider not only strictly equal pieces of codes, but also similar pieces of code. Such behaviorally similar code fragments are referred to as *simions* [10], in literature. They are different from Type 4 simple clones (described in related work), as the term clone is indicative of one derived from another while simians are usually developed independently.

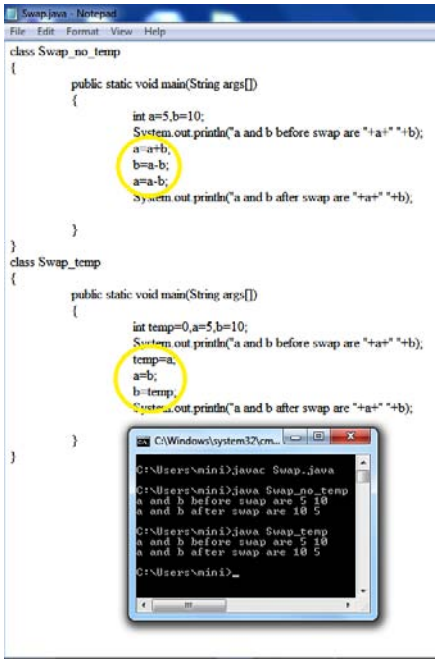

Figure 2. An example of behavior clone

The question that arises here is, do such codes possess syntactic similarities detectable by existing clone detection tools or not? Juergens et al. [11] demonstrated that it is highly unlikely for behaviorally similar code of independent origin to be syntactically similar. Such codes exist in real world and must be dealt with, since they pose maintenance threats. The significance of detecting run time behavior similarities further lies in unavailability and disguise of source code.

A program dependence graph is only an abstraction of program semantics, and nature a precise capture of program semantics. We may argue that it thus does not capture semantic similarity information precisely. Hence two or more code fragments with isomorphic program dependence graphs may or may not have similar behavior.We may conclude, behaviorally equal codes are behavioral clones. Behaviorally similar codes are simions. Semantically equivalent codes have isomorphic program dependence graphs and are called semantic clones, which may or may not overlap with behavior clones. It is assumed that all of these clones are developed independently and hence are different from Type 4 simple clones.

We now define hierarchical subclasses of semantic clones using the concept of program dependence graph [12]. A *program dependence graph* is a connected graph where nodes are entities and edges are relationships between entities. In Figure 3, we see graph S consisting of four entities (e_1, e_2, e_3, e_4), and four relationships (r_1, r_2, r_3, r_4) indicated by links among entities. A *relationship* is a meaningful physical or logical connection between two entities in a structure. One important relationship is the location of the interrelated entities. For example, we may be interested in entities found in the 'same method', 'same class', 'same file', 'same module', 'same sub-system' and so on. Other examples of relationships include 'message passing', 'inheritance', 'association', 'composition' among classes, or 'hyperlink' among web pages. Any other physical, semantic or syntactic relationship among entities may be included in a definition of a structure depending upon the context of similarity analysis.

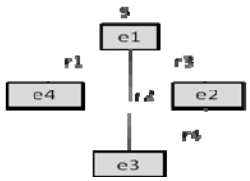

Figure 3. A program dependence graph

The following conditions define three orders of semantic clones, in a restrictive manner i.e. by definition, order 2 clones are order 1 and order 3 clones are order 2 and order 1 clones (Figure 4). For any two or more program dependence graphs,

- Their corresponding entities are in clone relation : Order 1 semantic clones
- Their program dependence graphs are isomorphic : Order 2 semantic clones

- Their corresponding relationships are same : Order 3 semantic clones

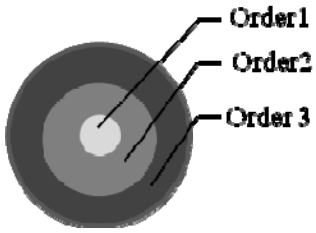

Figure4. Relation between various orders of semantic clones

In Figure 5, S1,S2, S3 and S5 are Order 1 clones because their corresponding entities e1a, e2a, e3a and e5a are in clone relation (we assume here that same shade fill in entity implies clone relation among entities). Note that S4 is not an Order 1 clone with S1, S2, S3 and S5 because entity e4a is not in clone relation with other entities (depicted by use of different shade fill). Had S4 been Order 1 clone, S1, S2, S4 and S5 would have been Order 2 clones too, since their connected graphs are isomorphic. Here, S3 violates this condition and hence is excluded from a higher order, thus, it stays in Order 1 only. Note that S5 has relation 'w' and not 'y' in the corresponding place like S1, S2 and S4. This leaves us with only S1 and S2 satisfying condition for Order 3. These orders can help us lay foundation for classifying high level clones (semantic and possibly behavior) clones more explicitly and help in visualizing the extent of similarity in higher level clones through a simple representation.

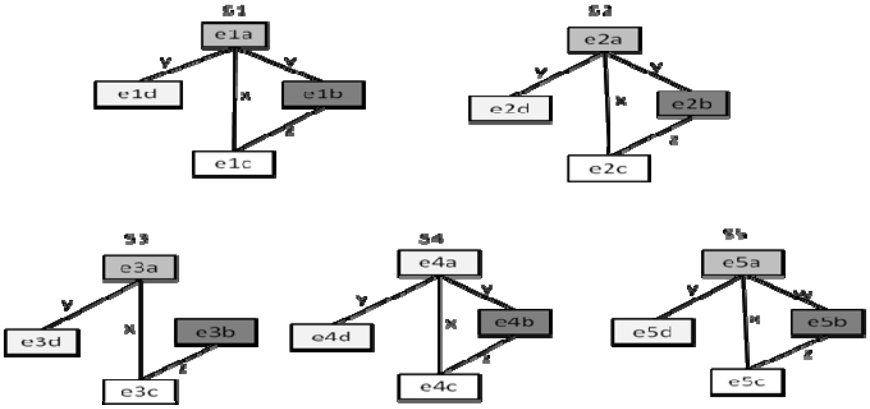

Figure 5. Orders of semantic clones

2.2. Concept Clones

In the example shown in Figure 6, the accessibility of model, view and controller among each other, for User module are analogous to those of Itinerary module i.e. if the view, controller and model of User module (named respectively as-home.jsp,UserServlet and UserModel) can access UserBean; then view, controller and model of Itinerary model (named respectively as-user_page.jsp, ItineraryServlet, ItineraryModel) can also access ItineraryBean. In the same manner, other interactions spread across MVC in the figure can be explained to be similar. Here, a concept clone (solid and dashed arrows) spans the Model View Architecture triad.

From the experience of Marcus [13], concept clones manifest themselves as higher level abstractions in problem or solution domain like an ADT (Abstract Data Type) list. Their detection is improved with internal documentation and program semantics. Design patterns and mental templates also fall under the category of concept clones.

The file structure of software system is examined with the aim to extract conceptual information embedded in source code elements and identifiers, to reduce the amount of source code an engineer needs to view, and also hint at possible relationships in the system which is not evident from system file organization or documentation. This is possible because the presence of concept clones implies that the two or more clone implementations had similar high level concept as their starting point.

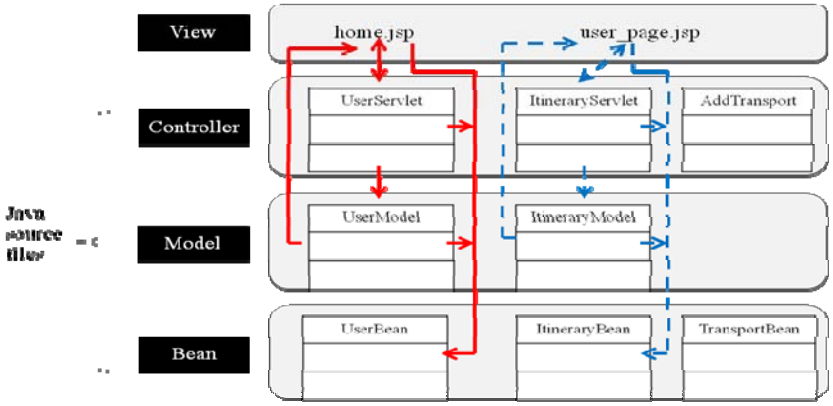

Figure 6. An example of concept clone

It is worth noting that the definition of high level clone is based on user's understanding of the system and thus may differ from user to user. It is thus not possible to detect these clones in fully automated manner without involving human interaction in the process. On the basis of concept clone detected, many reengineering decisions will be influenced, like using inheritance instead of cloned compilation unit. Ideally, concept clones would be merged into two or more classes during reengineering.

2.3. Collocated Simple Clones

A real life example of collocated simple clones is depicted in Figure 7. This is from a Java web application for travel itinerary. The user of this website wishes to manage hotel, transport and activity details. Here addHotel file has addHotel() method which is a simple clone of addTransport() method of addTransport file. Also, editHotel() method is similar to editTransport() method, and displayHotel() method is similar to displayTransport() method. This is true for all files in hotel, transport and activity modules. In effect, these modules bear a large granularity similarity i.e. module level. Thus, we abstract clones up the hierarchy from simple to high level clones. (Here, collocated simple clones).

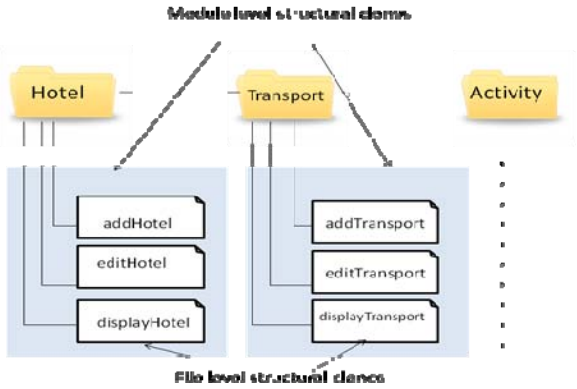

Figure 7. Example of collocated simple clone

Basit[14,15] visualizes collocated simple clones in Figure 8, as similar program structures that are formed by lower level, with similar code fragments (i.e., simple clones) at the bottom of such hierarchy. This concept of moving from lower level similarities to higher level similarities can be repeatedly applied, leading to the discovery of design concepts at higher levels of abstraction.

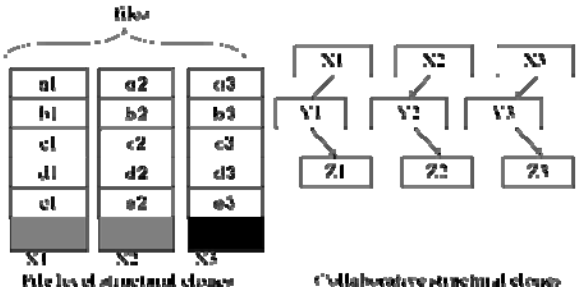

Figure 8. Collocated simple clones

2.4. UML Domain Model Clones

To illustrate this concept, we input a Java project to StarUML for reverse engineering and filter a package containing .java extension files for reverse engineering purpose. This results in a class diagram as shown above in Figure 9. It demonstrates class diagram level similarity; here all three classes have same data members ('con' and 'st'). In addition, ItineraryModel and Delegate classes have dbConSet() method, therefore a super class or interface may be constructed to abstract out cloned information in the UML class diagram. Similarly, other results of reverse engineering like ER diagrams and use case diagrams may manifest similarities. This information is useful to eliminate maintenance problems caused by unchecked domain model clones.

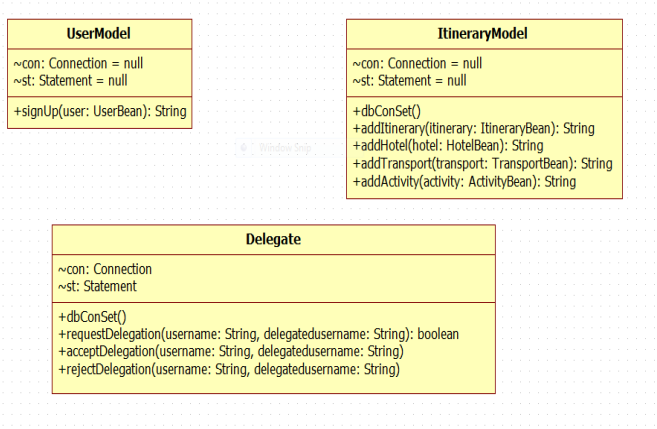

Figure 9. An example of UML domain model clone

It is likely that at some point, the overall model will contain duplicate fragments of sub models or model elements i.e. model clones [9].Clones in UML domain models can cause a problem during model based development, hence it is important to detect them. This is best possible through application of any of the reverse engineering tool, followed by domain expert's knowledge.

III. BENEFITS OF HIGH LEVEL CLONES' ANALYSIS

Recognizing higher level similarity can have significant value in the following areas:

Program understanding: When conceptual level similarity is identified, then it is much clearer to understand the basic concept behind a specific program, which is in line with a well known concept.

Program evolution: Sometimes a hierarchy of similarity exists in software i.e. certain low level clones form clusters and get reflected in design layer too. It is thus necessary to update design as and when the lower level implementation changes and vice versa, otherwise the update anomalies shall propagate the program hierarchy.

Reusable solution: In certain situations, the higher level similarity or structural clone is big enough to form a potential candidate for reusable solution. When similar design patterns are recovered through reverse engineering or similar conception recognition occurs through concept and domain analysis; these software artifacts can be put to use more than once in future.

Reengineering: A situation may be encountered where higher level similarity seems like an unnecessary burden to maintenance, because it does not improve efficiency of the program in any manner. This is poor design and could be improved by suitable refactoring [17]. Alternatively, overtly complex higher level design can also be simplified by reengineering.

IV. RELATED WORK

Basit et al.[15] observed that recurring patterns of simple clones (which he called structural clones), often indicated the presence of useful design level similarities. They used data mining techniques to detect such clones, and implemented the experimental results through a prototype tool.

Kwon describes behavioural clones to depict similar run time behaviour. These can be detected as: if any two programs having same Directed Acyclic Graph (DAG) modelling data dependency or same control flow dependency, they are behaviour clones of each other.Elmar et al call two pieces of code behaviourally equal, iff they have the same sets of input and output variables and are equal with respect to their function interpretation. Nagarajanet et al. analyze similarity in the control flow of programs. Jiang and Su present a technique to mine code fragments with similar input/output behaviour using random testing [16].

Kuhn et al. use an information retrieval technique which is language independent, called Latent Semantic Indexing (LSI) to cluster artefacts that use similar terms . Gabel defines code clones on the basis of program dependence graphs that represent control and data flow dependencies between statements and predicates [17].

Most of the researchers' work agrees on the presence and impact of high level clones in real software, but not much has been documented about their fundamental categorization and specific detection and treatment. The idea of high level clones being more abstract than simple clones is common to most work in this area. However, it is still an open issue whether the semi automated detection technique for such clones will define the subtype of high level clone type or it is predefined level of abstraction (like design, concept, model etc) which maps to corresponding high level clone subtype. Also, it is to be seen whether there can be a possible overlap in presence or even definition of these clone types.

V. CONCLUSION AND FUTURE WORK

Through this paper, we attempted to shed light on the need to give an unambiguous taxonomy of high levels of similarities present in software. We give consolidated definitions after studying relevant literature. This paper will provide an insight into possible classes and subclasses of abstract clones, however; the work is highly extendible as more clone types are discovered. Alternatively, it may be found that some of the definitions are slightly overlapping. We have an ongoing work on development of a fuzzy model for these four classes of high level clones where we visualize our results on Matlab's fuzzy tool box. We aim to extend our analysis to semi automated detection of each of these classes of clones, and accordingly provide refactoring guidelines for each of them.

REFERENCES

[1] Rainer Koschke, Ettore Merlo and Andrew Walenstein : Duplication, Redundancy, and Similarity in Software, in Dagstuhl Seminar Proceedings, 2007.

[2] William S. Evans , Christopher W. Fraser and Fei Ma : Clone detection via structural abstraction in Software quality journal Volume 17, Number 4, 2009.

[3] Cory Kasper and Michael W. Godfrey: Cloning considered harmful: considered harmful in Working Conference on Reverse Engineering '06, 2006.

[4] H. A. Basit and Stan Jarzabek: A Case for Structural Clones, International Workshop on Software Clones (IWSC), 2009.

[5] Kasper, C. and Godfrey M.W Toward a taxonomy of clones in source code: A Case study in Evolution of Large Scale Industrial Software Architecture, 2003.

[6] H. A. Basit, Usman Ali and Stan Jarzabek : Viewing Simple Clones from Structural Clones' perspective, in IWSC, Honolulu, 2011.

[7] Rainer Koschke: Begun the Software Clone War Has in SOFSEM software seminar, 2011.

[8] H. A. Basit, Damith C. Rajapakse and Stan Jarzabek : Structural Clones-Higher-level Similarity Patterns in Programs, SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, May, Lisbon, 2005.

[9] Nicolas Gold, Jens Krinke, Mark Harman,David Binkley: Proceedings of the 4th International Workshop on Software Clones IWSC '10, ACM, 2010.

[10] Mark Gabel, Lingxiao Jiang, Zhendong Su: Scalable detection of semantic clones in ICSE 2008, pages 321-330, 2008.

[11] E. Jürgens, F. Deissenboeck, and B. Hummel: Code Similarities Beyond Copy & Paste, in proceedings of CSMR, pages 78-87, 2010.

[12] H. A. Basit, S. Jarzabek : Detecting Higher-level Similarity Patterns in Programs, in proceedings of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, Lisbon, 2005.

[13] Marcus, A., and Maletic, J. I. : Identification of high-level concept clones in source code. in proceedings of the International Conference on Automated Software Engineering (ASE), pages 107-114, 2001.

[14] H. A. Basit and Stan Jarzabek :A Data Mining Approach for Detecting Higher-Level Clones in Software in IEEE Transactions On Software Engineering, Vol.35, No.4, 2009.

[15] H. A. Basit, Damith C. Rajapakse and Stan Jarzabek: Beyond Templates: A Study of Clones in the STL and Some General Implications published at Int. Conf. on Software Engineering(ICSE'05), 2005.

[16] Adrian Kuhn, Stephane Ducasse and Tudor Girba: Reverse Engineering with Semantic Clustering in proceedings of the 12th Working Conference on Reverse Engineering, IEEE Computer Society, 2005.

[17] Taeho Kwon and Zhendong Su: Modeling High-Level Behavior Patterns for Precise Similarity Analysis of Software, in the 11th IEEE Conference on Data Mining (ICDM), Vancouver, Canada, December, 2011.