IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 9, 2019, accepted April 10, 2019, date of publication April 29, 2019, date of current version May 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2913043

Evolutionary Perspective of Structural Clones in Software

JAWERIA KANWAL[1], ONAIZA MAQBOOL[1], HAMID ABDUL BASIT[2], AND MUDDASSAR AZAM SINDHU[1]

[1]Computer Science Department, Quaid-i-Azam University, Islamabad 45320, Pakistan
[2]Syed Babar Ali School of Science and Engineering, Lahore University of Management Sciences, Lahore 54792, Pakistan

Corresponding author: Jaweria Kanwal (kjaweria09@yahoo.com)

This work is supported by IGNITE Funding, Pakistan, grant no SRG-257.

ABSTRACT Cloning in software represents similar program structures having its own benefits and drawbacks. Proper clone analysis is required to exploit the benefits of clones. A study of software clone evolution serves the purpose of understanding the maintenance implications of clones, which leads to their appropriate management. Structural clones (recurring patterns of simple clones) represent design level similarities in software. Evolutionary characteristics of clones can assess the relevance of those clones for software developers and maintainers. Although the evolution of simple clones has been thoroughly studied by researchers, the evolution of structural clones is still to be explored. In this paper, we study the evolution of structural clones by performing a longitudinal study on multiple versions of five Java systems. To perform a systematic study of the structural clone evolution, we define structural clones and their evolution patterns in a formal notation. Our results show that structural clones are more likely to change inconsistently, however, less frequently than simple clones, whereas the lifetime of the structural clones is similar to that of the simple clones. Evolutionary characteristics of structural clones suggest that they require more attention in their management. Analysis of structural clone evolution reveals similar reasons for changes, and similar trends in evolution patterns, for all subject systems. These trends reveal evolutionary characteristics of structural clones that can help in devising appropriate strategies for managing them, hence devising better clone management systems.

INDEX TERMS Clone evolution, software clones, structural clones, software engineering, software evolution, software design.

I. INTRODUCTION

Copying an existing code fragment and reusing it in another part of software is a common practice in software development and generally known as code cloning. There are many reasons of cloning. They may be intentional, e.g., reuse of tested code, or unintentional, e.g., implementation of similar features during development [1]. Previous research shows that code clones commonly exist in almost all kind of software systems as similar solutions are repeatedly used to solve similar problems. Certain programming styles e.g. architecture-centric and pattern-driven development also support similar program solutions to maintain the same standard. Empirical studies show that generally software contains 9%-17% of clones [1] and in some specific types of software systems e.g. java buffer library, clones cover around 50% of the system code [2]. Different empirical studies to analyze code clones show that code clones may be both useful and harmful for software maintenance [3]. There are some advantages of using code clones during development. Cloning of existing code provides reuse benefit i.e. it saves time and effort for devising new solutions for similar problems [4]. Moreover for implementation of similar features, use of cloning approach facilitates in independent evolution of the code. However, sometimes clones cause additional maintenance effort by introducing inconsistent change in similar code fragments [5].

Regardless of their benefits and drawbacks, understanding software cloning is very important for better clone management. A developer should understand cloning so that the change impact of clones for a particular maintenance task can be evaluated. For effective clone management, different

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

aspects of clones need to be addressed which include detection of clones at different granularity levels, understanding clones from a historical perspective. A deeper understanding of how clones are handled during the evolution of a software system is helpful in managing clones properly as it suggests the appropriate time and methods to deal with clones [6]. There is a need to devise methods and tools so that positive effects of cloning can be exploited and risks can be minimized [7]. For these reasons, in recent years, clone detection and analysis has received considerable attention from the software engineering research community [1].

There are two broad categories of software clones, simple clones and structural clones. Simple code clones are textually or syntactically similar code fragments in software programs [1]. Existing tools for clone detection typically report a large number of code clones. It becomes difficult for a developer to understand the cloning in a system at a higher level, thus structural clones are introduced. Structural clones are recurring patterns of simple clones in software [8]. They represent program similarity at the code level as well as at the design level. The bigger picture of similarity represented by structural clones leads to design similarity such as domain specific design solutions of a system [9], [10]. Structural clones in software may represent that software systems contain design level similarities [11]. Some software systems such as Java Buffer library's code contain at least 68% clones in cloned classes or class methods which is the result of design similarities. Kapser and Godfrey [3] list several patterns of cloning that are used in real software systems and argue that clones can be a reasonable design decision.

Identification of clones that might be interesting for developers during software maintenance is an important research topic in clone research. Empirical studies on clone evolution has shown that history of clones helps in identifying and filtering interesting clones regarding maintenance of software [12], [13]. It also helps in categorizing clones according to their evolutionary behavior and provides useful information of different aspects of clones [14]. For example, change frequency of clones is a metric that helps in identifying interestingness of clones i.e. the clones that are changing frequently but still remain in the system for long time may be more interesting for maintainer than those that disappear earlier. This information is helpful for the developer during software maintenance, as a developer may handle frequently changed clones differently than other clones in the system for a particular maintenance task. These studies help in devising tools for clone management. In the literature, there are various tools such as CReN [15] and CSeR [16], that support developers in simple clone management [17], [18].

Although there have been studies for simple clones, but so far there is no available work on studying the evolution of structural clones. Evolutionary characteristics of structural clones will help in determining the relevance of structural clones for software maintenance. A deeper understanding of how structural clones change during a software's lifetime will help in determining what kinds of modification and refactoring support are required for managing structural clones effectively. Study of structural clone evolution leads to devising better clone management systems. In our previous research we performed an initial study on structural clone evolution [19]. In this paper, we enhance our study by defining terminology related to structural clones and structural clone evolution formally so that evolution of structural clones can be analyzed more effectively. Further we present our approach in detail, conduct experiments on five subject systems and provide detailed discussion of the results. Our major contributions in this paper are as follows:

1) We present a formal definition of structural clones. As there is no standard definition of structural clones and researchers have described them informally in different perspectives, there is need to define them formally so that the framework for studying their evolution can be established.

2) We define clone evolution patterns for structural clones. Clone evolution patterns exist for simple clones, however for structural clones they need to be redefined as structural clones represent recurring patterns of simple clones and logical relations among them.

3) We present a detailed study of structural clone evolution by considering multiple versions of five well-known subject systems. We study the evolution of structural clones by answering the following research questions:

RQ1: What are the characteristics of structural clones compared to simple clones in terms of clone lifetime?

RQ2: How consistently and frequently structural clones change than simple clones?

RQ3: How structural clone classes change between the versions in evolving software?

RQ4: How evolution of structural clones differs from the evolution of simple clones?

Rest of this paper is organized as follows. Section 2 discusses literature survey on clone evolution and structural clones. Section 3 presents formal definitions related to software clones. Section 4 describes clone evolution patterns formally. Section 5 presents our proposed approach. Section 6 reports the results of clone evolution and analyzes them in detail. Section 7 presents maintenance implications of structural clone evolution and finally section 8 concludes the results.

II. LITERATURE SURVEY

There are a number of good literature surveys on software clones. Roy et al. presented the first state of the art survey on clone management [1]. Koschke [20] presented a brief discussion and open questions on different aspects of clone research including clone detection, evolution and cause-effect of clones. A systematic review on clone evolution is also presented in [21] which discusses various studies performed to understand clone evolution. In the following we discuss literature on structural clones and work that is particularly important in the areas of clone evolution and most closely related to our study.

Empirical studies on code clone evolution show that evolutionary aspects of clones such as clone life, consistent/inconsistent change and frequency of change cannot be revealed through analysis of single version [22], [23]. Evolution of simple code clones has been studied by different researchers. One of the earliest studies on code clones evolution in a systematic way is done by Kim *et al.* [12]. They performed a longitudinal study on the evolution of code clones and defined a formal model for studying clone evolution. Changes in clone groups are studied through clone genealogies that represent the changes (or evolution patterns) in clone groups in various versions of software system. Analysis of clone genealogies showed that many clones disappeared within few check ins which reveals that aggressive refactoring of clones is not appropriate for all kinds of clones. Only 36%-38% of clone genealogies are changed consistently and remain in the system for a long time. These kind of clones cannot be generally refactored through popular refactoring methods. They concluded that all clones are not refactorable and hence remain in the system till the last release of the software.

Aversano *et al.* [14] studied clone evolution at clone instance (code fragment) level. They analyzed changes in clones through CVS transactions and analyzed the clones that evolve inconsistently in more detail at different granularity levels such as block, method and class levels. Concept of late propagation was defined for the first time which is considered an interesting pattern in clone evolution research. This pattern described that some clones change consistently but not in the same revision i.e. some of the clone instances of a clone group change in earlier revisions while others change in later revisions. They showed that as a result of late propagation, some clone groups disappeared in a revision and then reappeared after few revisions. Analysis of clone genealogies revealed that there were two main reasons of this pattern. Some clone instances were removed in a version by unintentional refactoring of clones and then in next versions it was realized that cloning was unavoidable. The other reason was inconsistent change where developers change some instances and leave others unchanged.

Evolution of different types of clones such as Type 1, Type 2 and Type 3 clones has also been studied. These studies showed that different types of clones exhibit different evolution characteristics. Bazrafshan [13] studied the evolution of different types of code clones. They called the Type 1 clones as identical clones and Type2 and Type3 clones as near-miss clones. A comparison was performed between the two categories of clones and it was observed which clone type was more long-lived, change-prone and had an impact on clone ratio. Life time analysis of different types of clones showed that near-miss clones were generally more long-lived than identical clones. It was also observed that near-miss clones are changed more frequently and more inconsistently than identical clones. Analysis of clone type interchangeability showed that there were some identical clones that were converted into near-miss clones during system evolution in majority of the studied systems. ArgoUML was the only system where some near-miss clones were converted into identical clones.

In [22], evolution of Type 3 clones is studied to understand the evolutionary characteristics of Type 3 clones as compared to other clone types. Study showed that there is no significant difference among life time of the three clone types. Evolution study of clone types revealed that different clone types are interchangeable i.e. in some cases, Type 3 clones were converted into Type 1 and Type 2 clones during evolution and vice versa. Type 3 clones are less stable because they changed more frequently and more inconsistently than Type 1 and Type 2 clones during system evolution.

Mondal *et al.* [24] studied the impact of different evolution patterns for Type 3 clones. Bug proneness of inconsistent changes specially in late propagation of changes was observed. Results show that only 10% late propagations introduce bugs in the system.

In a recent study [25], evolution of clones is studied at the level of micro clones. Micro clones are the code clones that are smaller in size than the regular clones detected by conventional clone detection tools. Size of regular clones is usually considered more than 5 lines. In the area of clone research, small sized clones are not considered as interesting for maintainers, especially for refactoring. Evolution study of micro clones reveals that micro clones are consistently updated by maintainers during evolution, which shows that detection and management of micro clones can help in system maintenance. Analysis shows that 23% changes in micro clones are consistent changes and 80% of the total consistent updates of the system occur in micro clones which is significantly higher than regular clones which is only 16%. Proportion of consistent modifications were higher than percentage of consistent additions or deletions of code in the studied systems. Micro clones that consist of one or two lines are changed more consistently than larger micro clones.

Zhang *et al.* [26] proposed an approach to predict the need for making consistent change in clones within a clone group at the time when changes have been made to one of its clones. The experimental results showed that the prediction models have reasonable rate: they have good precision and recall rates for two large piece of repositories, and reasonable precision and recall rates for the smallest repository. In addition, all three sets of attributes have strong impact on recall rates of the prediction models.

Measuring the stability of code clones is also very important in software evolution. There are different studies on how changes in code clones impact the code stability of software. Mondal *et al.* [23] proposed a new method for measuring the stability of code. A new metric called "change dispersion" is defined to observe the code stability. This metric measures the number of entities that are impacted by a change. Results show that the more the number of entities are impacted by a change, the more maintenance effort is spent on that code. Thus more the code is unstable.

In [27] Mondal et al. compared the stability of cloned and non-cloned parts in a software. They used eight code stability metrics and seven different methodologies in a common framework to analyze the code stability in a software. Experiments were performed on 12 subject systems from three programming languages. Change impact was observed at method level granularity. Results show that cloned code is more change-prone than non-cloned code in the studied software systems. Experiments are performed on different types of clones and it is observed that Type1 and Type3 clones are more unstable than Type2 clones. Results of almost all the stability metrics have shown that cloned code is more unstable than non-cloned code i.e. changes in cloned code are more dispersed than non-cloned code. Analysis of software with different programming languages showed that Java and C language systems are more unstable than C# in terms of clone stability.

Clones have also been analyzed for program comprehension. Elevating the level of cloning to higher granularity levels facilitates in software understanding, maintenance and evolution [8]. In the following, we will discuss a few studies on higher level clones.

In [28], authors extracted similar business and programming rules in the software and named them logical clones. These logical clones contain simple clones and functional clusters (consisting of methods), files, data objects as entities and logical relations such as method calls, inheritance as relations among entities. A functional cluster contains methods with similar functionality. Logical relations between software entities e.g. methods, files, are extracted and models of software are built based on these relationships. They showed that logical clones are beneficial for understanding similarity of business rules in software.

Syntactical pattern clones [29] represent the clones that share similar syntactic context e.g. Inheritance. These are the grouping of simple code clones that are also similar in some kind of syntactic relation. For example, in a Java system two clone classes belong to a group of Java classes that are sibling of each other (extended from same class) or implementing same interface. Identification of syntactic similarities helps the maintainer to manage clones according to their syntactic context.

Basit and Jarzabek [8] observed that recurring patterns of simple code clones present higher-level similarities in software. These higher-level similarities are called structural clones by them. Structural clones show a broader picture of code similarities than simple clones alone. A technique is proposed using Association Rule Mining to find these similarity patterns in software. The technique is implemented in a tool called CloneMiner to find structural clones automatically. Structural clones at level of files and methods are found in different software systems and analyzed. Analysis of different systems revealed that structural clones represent some software design concepts. Analysis of Java Buffer Library showed that the identified structural clones represent design concepts that can be automated through automatic code generation techniques such as use of XVCL technique.

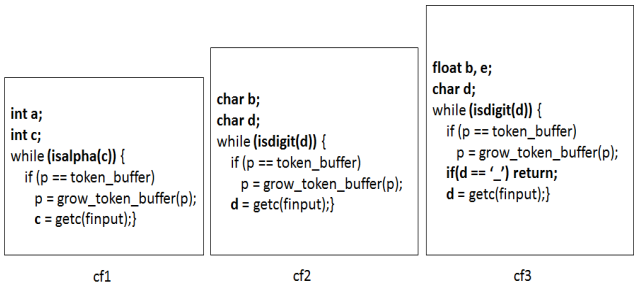

FIGURE 1. An example of simple code clones.

In [9], analysis of structural clones showed that they indicated application domain concepts or system specific design. The formation of structural clones allows to view cloning at higher granularity level which helps in understanding clones for further maintenance tasks. A detailed analysis of distribution of simple clones and different types of structural clones is performed to measure how frequent are structural clones in various software systems and how these clones help in understanding software design and clone management [11]. Experiments were performed on eleven software systems and results showed that more than 50% of simple clones are present in structural clones. Analysis of the structural clones revealed that structural clones help in understanding software clones which leads to their management. So far, there has been no study on structural clones evolution.

III. TOWARDS A FORMAL DEFINITION OF SOFTWARE CLONES

Software clones can be categorized into two broad categories according to their level of granularity, Simple code clones and structural clones. In this section, we provide formal definition of both categories.

A. SIMPLE CODE CLONES

A set of code fragments that are similar according to some similarity criteria are called clones of each other. A code fragment cf is a 4-tuple $\langle S, L, M, F\rangle$ consisting of a chunk of source code starting at line S with the following L lines of code in a method M which is declared in file F [30].

1) CLONE PAIR

Two code fragments $cf_1 \wedge cf_2$ are called clones of each other if they are similar according to some similarity criteria. A *Clone pair* (CP) can be defined as $CP = \{cf_1, cf_2\}$ such that $sim(cf_1, cf_2) > k$ *where* k is a similarity threshold which represents similarity of code fragments. The similarity function $sim(cf_1, cf_2)$ can measure textual, semantic or syntactic similarity [31] depending upon the clone detection technique used by the clone detection tools.

2) CLONE CLASS

Contains set of code fragments cf which are similar to each other. A *Clone class* (CC) can be defined as

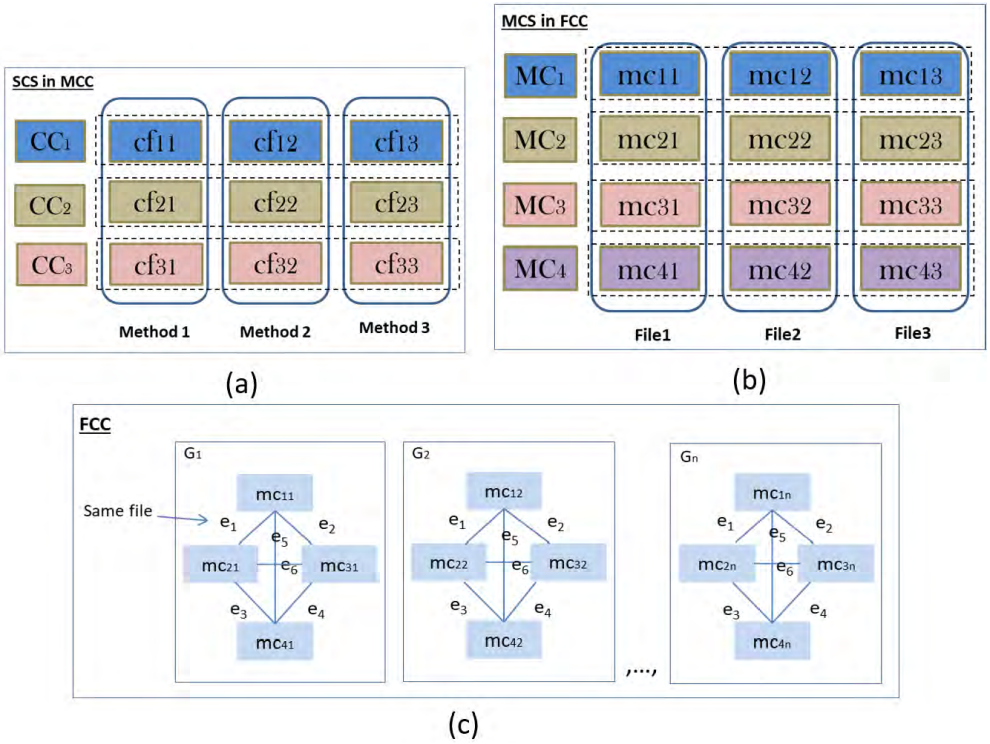

FIGURE 2. Structural clones (a) simple clones in methods (MCC) (b) method clones in files (FCC) (c) graphical representation of FCC.

$CC = \{cf_1, cf_2, cf_3, \ldots, cf_n\}$ such that for any pair cf_i, cf_j, $sim(cf_i, cf_j) > k$, $1 \leq i, j \leq n \wedge i \neq j$.

In this paper, clone pair and clone class will be referred to as simple clones. An example of simple clones is shown in Figure 1 where three code fragments cf_1, cf_2, cf_3 are similar to each other and are clone instances of a clone class.

B. STRUCTURAL CLONES

The concept of structural clones was introduced to understand cloning at a higher level than simple clones [8]. Structural clones are the recurring patterns of simple clones in software. Structural clones are formed by grouping clone classes (two or more) that are participating in those recurring patterns. For example, a group of clone classes whose clone instances are repeatedly occurring in same files, form a structural clone class [8]. There are various types of structural clones discussed in literature [8]–[10] ranging from small software entities e.g. method level clones to bigger entities like component level clones. In this section we discuss various examples to understand the concept of structural clones and their usefulness in software maintenance.

The most basic type of structural clones discussed in literature are containment based structural clones. Detection of containment based structural clones is supported by the CloneMiner [8] tool. Table 1 represents various types of containment based structural clones. Column1 shows types of clone classes and column2 shows types of recurring patterns of clones in software entities.

The simplest type of structural clones are Method Clone Classes (*MCC*) as shown in Figure 2(a). Recurring patterns

TABLE 1. Various types of software clones detected by CloneMiner.

Clone Classes	Clone Structures
Simple Clone Class (SCC)	Simple Clone Structures (SCS) Across Methods/ Across Files
Method Clone Class (MCC)	Method Clone Structures (MCS) Across Files
File Clone Class (FCC)	File Clone Structures (FCS) Across Directories/Within Directoris

of Simple Clone Classes (*SCC*) are called Simple Clone Structures (*SCS*). The methods that contain *SCS* and fulfill a defined threshold (percentage of *LOC* of *SCS*/*LOC* of methods) of *SCS* are called *MCC*. Generic representation of *MCC* is given in Figure 2(a) where clone instances of three simple clone classes CC_1, CC_2 *and* CC_3 are repeatedly occurring across three methods forming three *SCS* i.e. $\{cf_{11}, cf_{21}, cf_{31}\}$, $\{cf_{12}, cf_{22}, cf_{32}\}$ and $\{cf_{13}, cf_{23}, cf_{33}\}$. An example of *MCC* from JHotDraw consists of variations of init( ) method in four different files *DrawApplet.java, NetApplet.java, PertApplet.java* and *SVGApplet.java*. Clone instances of three simple clone classes repeatedly occur in these methods. To understand the concept of *SCS* in MCC through a real world example, Figure 3 shows source code of init() method of two files SVGApplet.java and PertApplet.java which consist of clone instances of three different simple clone classes (Source code of these files are not shown here to save space). In this example, three clone instances in init() method of SVGApplet.java form an SCS_1 and three clone instances in init() method of PertApplet.java form SCS_2. Similarly, clones in init() method of DrawAp-

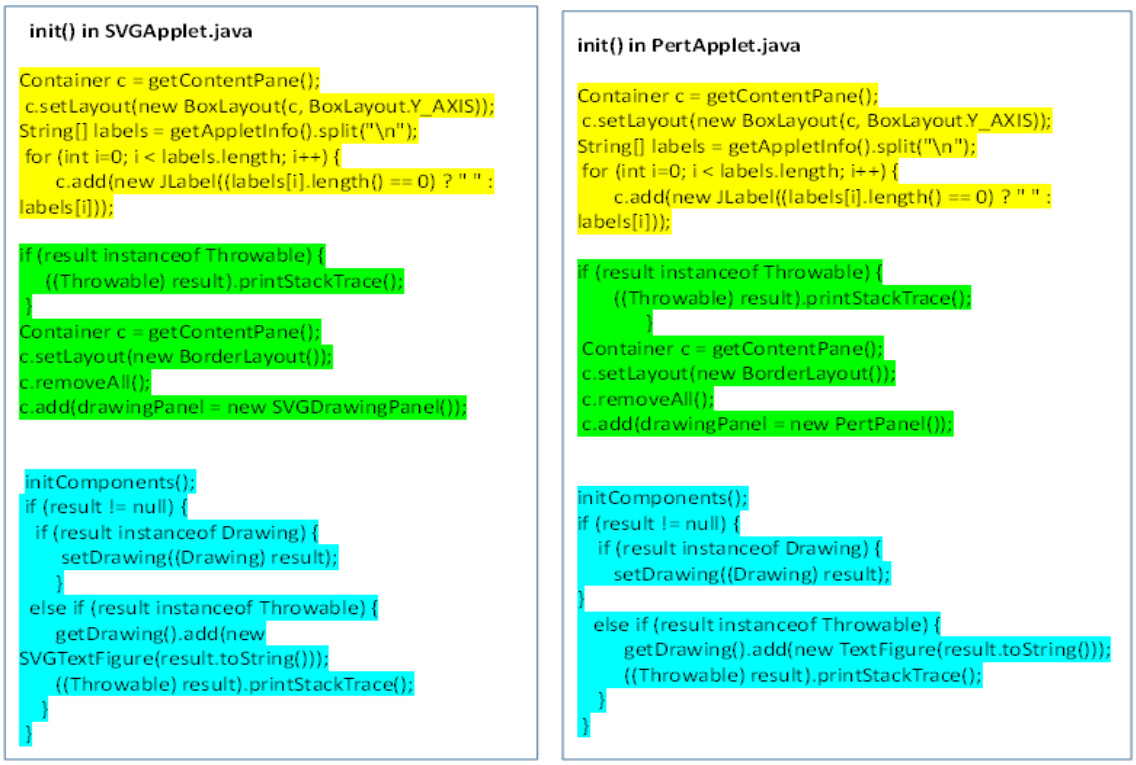

FIGURE 3. An example of SCS (Simple Clone Structures) in MCC (Method Clone Classes).

plet.java and NetApplet.java also form SCS_3 and SCS_4 which form an *MCC*.

These types of structural clones are beneficial in change impact analysis because they represent cloning at method level which is more manful thn code frg. Thus a developer knows all the simple clones present in the methods of an MCC which will help in understanding the impact of change at a broader level.

Another type of containment based structural clones are File Clone Classes (*FCC*). Recurring patterns of Method Clone Classes (*MCC*) in files are called Method Clone Structures (*MCS*) as shown in Figure 2(b). The files that contain MCS and fulfill a defined threshold of *MCS* are called File Clone Classes (*FCC*). Generic representation of *FCC* is shown in Figure 2(c). An example of *FCC* from JHotDraw 7.2 contains two files, SVGPathFigure and ODGPathFigure in package 'org.jhotdraw.samples.odg.figures'. The files share 11 cloned methods named *getPath( ), flattenTransform( ), restoreTransformTo( ), contains( ), setBounds( ), getDrawingArea( ) getTransformRestoreData( ), handleMouseClick( ), getActions( ), createHandles( ) and transform( )*. The reason of cloning in these files is the implementation of similar functionalities for two different types of figures *SVGFigure* and *ODGFigure*. Further analysis shows that these files are also similar in other perspectives such as both files extend the same abstract class *AbstractAttributedCompositeFigure* and implement same interface *ODGFigure*. Analysis of other structural clones shows that containment based structural clones often share other relationships also e.g. same inheritance relation or same method call. *FCC* can be helpful in software understanding and taking refactoring decisions at a higher level e.g. applying clone refactoring at file level instead of method level where needed. Directory Clone Classes (*DCC*) are the largest type of containment based structural clones. Recurring patterns of File Clone Classes (*FCC*) in directories are called File Clone Structures (*FCS*). Directories that contain *FCS* and fulfill a defined threshold of *FCS* are called *DCC*. An example of *DCC* is shown in Table 2. Recurring pattern of four *FCC* (File Clone Classes) exist in two directories *odg* and *svg*. Each *FCC* contains two similar files e.g. FCC_1 contains '/org/jhotdraw/samples/odg/ODGView.java' and '/org/jhotdraw/samples/svg/SVGView.java'. Examples of directory clones show that containment based clones also reveal system design and can be helpful in software understanding and design recovery.

TABLE 2. An example of directory level clones.

File Name	File clone id	directory Name
/org/jhotdraw/samples/odg/ODGView.java	FCC1	odg
/org/jhotdraw/samples/odg/ODGAttributeKeys.java	FCC2	odg
/org/jhotdraw/samples/odg/ODGConstants.java	FCC3	odg
/org/jhotdraw/samples/odg/PathTool.java	FCC4	odg
/org/jhotdraw/samples/svg/SVGView.java	FCC1	svg
/org/jhotdraw/samples/svg/SVGAttributeKeys.java	FCC2	svg
/org/jhotdraw/samples/svg/SVGConstants.java	FCC3	svg
/org/jhotdraw/samples/svg/PathTool.java	FCC4	svg

In the following, we will discuss a higher level of structural clones where a set of File Clone Classes (*FCC*) have a calling relation with each other. These structural clones present an

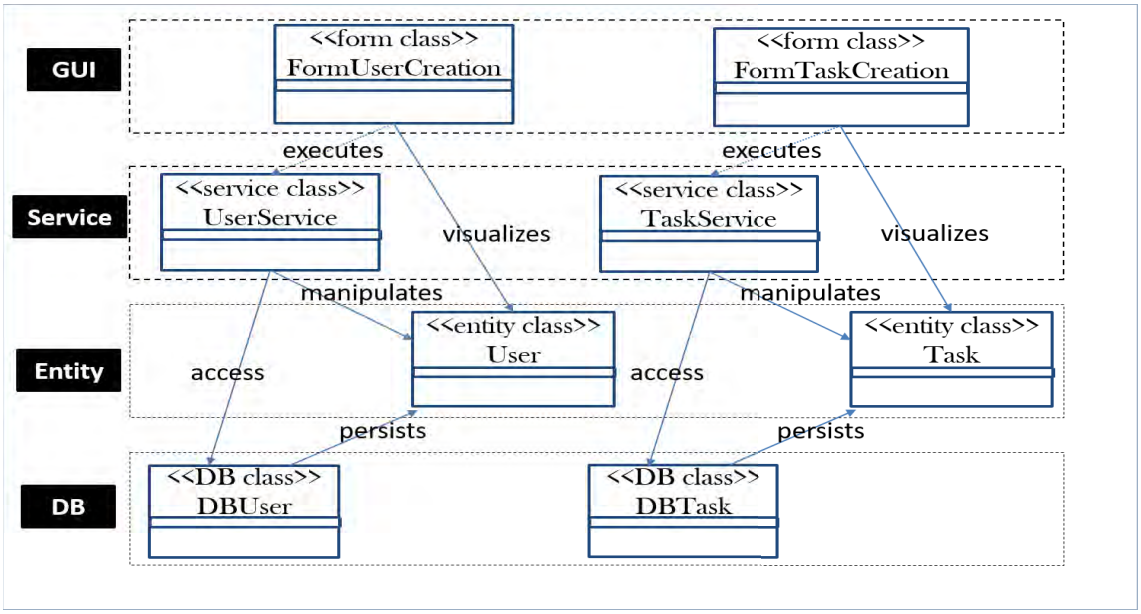

FIGURE 4. An example of structural clones (FCC) of collaborating classes.

example of component level cloning in the software where clone instances consist of recurring patterns of collaborating File Clone Classes across different layers of software. Each *FCC* consists of object classes where classes are clones of each other.

Figure 4 shows an example of structural clone of collaborating classes across different layers of software. This is found in a real C# system of an industrial company (SES Systems Pte Ltd) [9]. This structural clone contains two instances. Each instance consists of four classes calling each other with the same pattern. These clones arose from the following situation:

The system consists of more than 20 domain entities e.g. User, Task. Various operations e.g. Create() for these entities are designed and implemented in the same manner, forming clones of each other. Each box in Figure 4 represents a File Clone Class (*FCC*) consisting of a number of classes implementing similar concepts/operations for different entities. As a result when a certain operation is executed for any entity, methods of the classes at GUI layer execute functionality of classes of Service layer and further, classes at service layer access data entities at database layer. This type of structural clone depicts a pattern of collaborating classes across GUI, service and database layers.

Another example of a structural clone of collaborating classes across layers is found in JHotDraw V. 7.1 as shown in Figure 5. Structural clone consists of two File Clone Classes. FCC_1 contains three Java classes. i.e. NetApplet.java, PertApplet.java and DrawApplet.java. FCC_2 contains NetFactory.java, PertFactory.java, and DrawFigureFactory.java. Methods of Java classes in FCC_1 are calling methods of Java classes in FCC_2 and making a structural

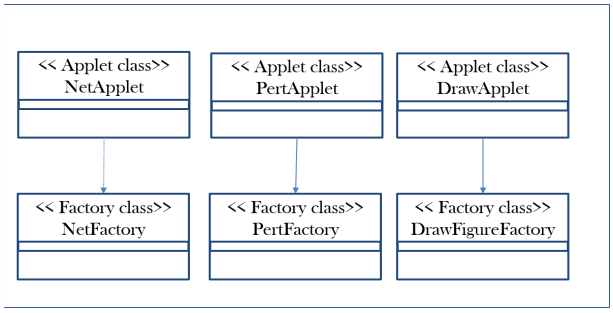

FIGURE 5. An example of file level structural clone (FCC) in JHotDraw.

clone of collaborating classes consisting of three instances. Each instance consists of two Java classes with a method call relation between them. These clones arose from the following situation:

JHotDraw is a Java drawing framework for developing graphics applications. To read and write objects of NetApplet, it requires parameters of NetFactory. Methods of three Java classes i.e. NetApplet.java, PertApplet.java and DrawApplet.java are calling corresponding methods in files NetFactory.java, PertFactory.java, and DrawFigureFactory.java. All the methods also use objects of NanoXMLDOMOInput and NanoXMLDOMutput to read and write data respectively.

As we discussed in Section 1, clones offer some benefits such as reuse of existing code. Above examples show that structural clones present a higher level similarity of code that is why reuse approach can be useful at higher level i.e. reuse of design. Structural clones can help in understanding and recovering system design through implementation. Moreover, structural clones also help in better software refactoring

than when only simple clones are considered because structural clones identify places where high level refactoring can be applied. As shown in Figure 5, three Java classes i.e. NetApplet.java, PertApplet.java and DrawApplet.java are clones of each other. A developer will consider refactoring of Java classes instead of clone methods of these classes (as in simple clones, refactoring can be applied at the level of code fragments or methods only) which may be more beneficial in terms of maintenance cost.

For structural clones, no formal definition has been provided. In the following, we provide formal definition of structural clones.

FORMAL DEFINITION OF STRUCTURAL CLONES

A structural clone, as shown in Figure 2(c) is a set of graphs G_1 to G_n where nodes of a graph G_i, $1 \leq i, \leq n$, are clone instances (of simple clones or containment based structural clones) and edges are relations between them e.g. same file. Each G_i represents a structural clone instance, a pair (G_i, G_j) is called a structural clone pair. In this paper, the terms structural clones, structural clone pair and structural clone class are used interchangeably.

A Structural Clone Class (SCC) can be defined as:

$SCC = \{G_1, G_2, \ldots, G_n\}$ *such that* $sim(G_i, G_j) > k$, $1 \leq i,j \leq n \wedge i \neq j$. G_i contains a set of nodes $N = \{cu_1, cu_2, cu_3, \ldots, cu_n\}$ *where* $cu_i \in \{CU_1, CU_2, \ldots, CU_n\}$, $1 \leq i \leq n$ and CU represent a Clone Unit which may be a simple clone class CC or containment based structural clone class (for example, in case of simple clones, $N = \{cf_1, cf_2, cf_3, \ldots, cf_n\}$ $cf_i \in \{CC_1, CC_2, \ldots, CC_n\}$, $1 \leq i \leq n$ and a set of edges $E = \{e_1, e_2, e_3, \ldots, e_m\}$ where $E \subseteq \{N \times N\}$ *such that* $e_i = (cf_p, cf_q)$, $1 \leq i \leq m \wedge p \neq q$. In case of structural clones, edges represents relationships among software entities e.g. inheritance, method call, containment. A set of such relationships are denoted as R. In containment based structural clones, a graph edge will be an element of R e.g. same file, as shown in Figure 2(c) whereas, in general, graph edges may be a subset of R.

As structural clones are represented through graphs, their similarity is a case of graph similarity. Before defining the structural clone formally, we will discuss graph similarity. Completely similar graphs are called isomorphic graphs. Two graphs are said to be isomorphic which contain the same number of graph vertices connected in the same way. G_1 and G_2 are said to be isomorphic if there is a permutation p of N such that $\{f_i, f_j\}$is in the set of graph edges $E(G_1) \iff \{p(f_i), p(f_j)\}$ is in the set of graph edges $E(G_2)$.

Isomorphic graphs represent identical (completely similar) structural clones. An example of an identical structural clone is given in Figure 2(c). However, structural clones are not necessarily completely isomorphic graphs. Similarity of graphs should fulfill some defined thresholds/criteria k. For example, consider two graphs G_1 and G_2 with graph vertices N_1 and N_2 such that $N_1 = \{cu_{11}, cu_{12}, cu_{13}, \ldots.cu_{1p}\}$ *where* $cu_{1i} \in \{CU_1, CU_2, \ldots CU_n\}$, $1 \leq i \leq p$ and $N_2 = \{cu_{21}, cu_{22}, cu_{23}, \ldots.cu_{2m}\}$ *where* $cu_{2j} \in \{CU_1, CU_2, \ldots CU_n\}$, $1 \leq j \leq m$.

G_1 and G_2 are said to be similar when (cu_{1i}, cu_{2j}) is in the set of graph edges $E_1(G_1) \iff (cu_{1i}, cu_{2j})$ is in the set of graph edges $E_2(G_2)$ and for node cu_{1i} *in* N_1, there is a node cu_{2j} *in* N_2 such that cu_{1i} and cu_{2j} are clone instances of same clone class and number of such mappings is defined by a threshold. Edges may be directed or non-directed depending upon the kind of relation.

Definition of structural clones covers all types of structural clones but containment based structural clones have been mostly discussed and analyzed in the literature [8] [11]. We discussed some examples of containment based structural clones in the beginning of this section. Graph representation of containment based structural clones is similar to the one shown in Figure 2(c), where all nodes are connected to each other and all edges represent only one relation i.e. same file.

IV. EVOLUTION OF SOFTWARE CLONES

To discuss clone evolution, it is important to first define the required terminology. For the purpose of our study, a software system $S = \{F_1, ..., F_n\}$ consists of a set of n source code files F_i, $1 \leq i \leq n$. A system exists in multiple versions V where the complete system can be retrieved for all versions V:

$$S(V) = \{FV_1, FV_2, ..., FV_n\} \tag{1}$$

is the system in versions V where $V_1, V_2, \ldots, V_n$ denotes versions of a system S. The differences between two versions V_1 and V_2 of a system can be identified by a set of changes: Let $D(V_1, V_2)$ denote the set of changes $\{d_1, ..., d_k\}$ between S(V1) and S(V2). Evolution of a software system is the study of changes between its different versions [30].

Evolution of software clones is studied through clone genealogies [12]. A clone genealogy is a directed acyclic graph that connects a clone class of a particular version with the corresponding clone classes in the next version. An example of clone genealogy is given in Figure 6. Clone genealogies are generated from clone classes across the versions based on clone evolution patterns. Clone evolution patterns are the changes a clone class went through from version V_n to the next version V_{n+1}. CC_n represents a clone class in V_n , CC_{n+1} represents its counterpart inV_{n+1}such that there exist a cloning relation between CC_n and CC_{n+1}. $CC(V_n)$ represents set of all clones classes in V_n and $CC(V_{n+1})$ represents set of all clone classes in V_{n+1}.

A. CLONE EVOLUTION PATTERNS

A clone evolution pattern represents a kind of change that a code clone class went through from one version V_n to a subsequent version V_{n+1}. Evolution patterns for simple clones are described in [12].

1) EVOLUTION PATTERNS FOR SIMPLE CLONES

Same: None of the clone fragments of a clone class in V_{n+1} was changed from V_n. For all $cf_i \in CC_n \wedge$ for all $cf_j \in CC_{n+1}$, $1 \leq i \leq Num(CC_n) \wedge 1 \leq j \leq Num(CC_{n+1})$

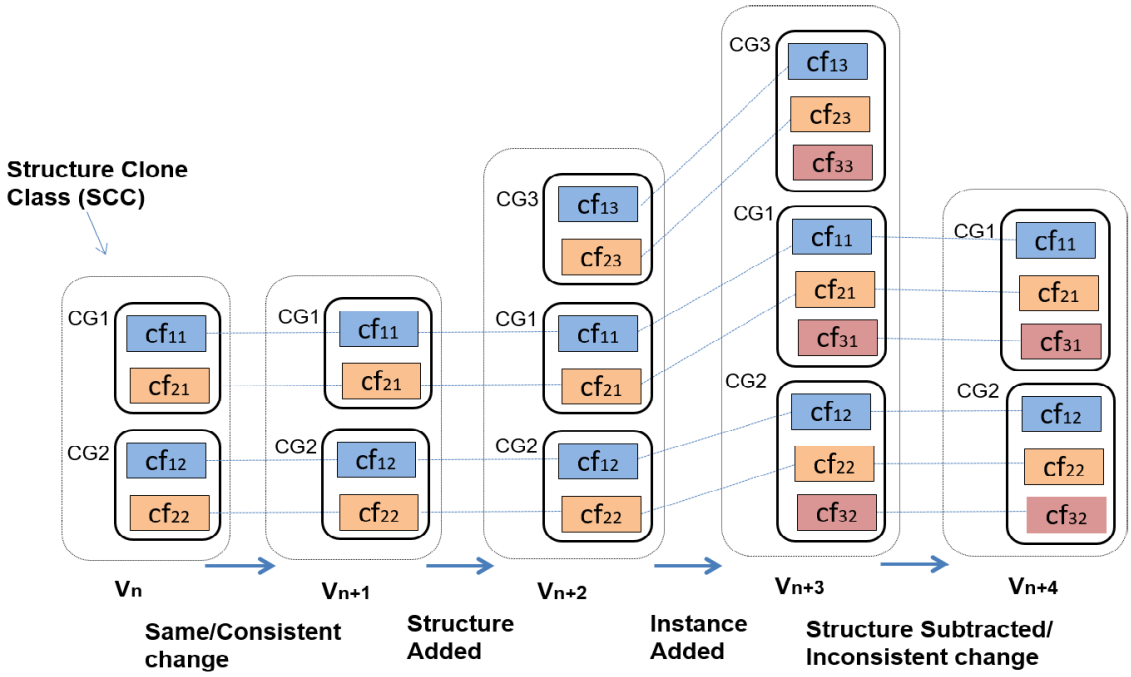

FIGURE 6. An example of structural clone evolution genealogy.

$CC_n.cf_i = CC_{n+1}.cf_j$ *for all* $i = j \wedge Num(CC_n) = Num(CC_{n+1})$ *where* $Num(CC)$ represents the number of code fragments in clone class CC.

Add: At least one code fragment in V_{n+1} is newly added.

$CC_{n+1} = CC_n \cup \{cf_j\}$ *where* cf_j represents a new code fragment $\wedge\ q \geq j \geq 1$ where q is a any finite number.

Subtract: At least one code fragment from V_n does not appear in V_{n+1}.

$CC_{n+1} = CC_n - \{cf_i\}$ *where* cf_i is a code fragment of $CC_n \wedge q \geq i \geq 1$

New: A new clone class CC_{n+1} is introduced in V_{n+1}.

$$CC(V_{n+1}) = CC(V_n) \cup \{CC_{n+1}\}$$

Removed: A clone class from V_n does not appear in V_{n+1}.

$$CC(V_{n+1}) = CC(V_n) - \{CC_n\}$$

In order to define the following two clone evolution patterns formally, we introduce the change function ϕ for code fragments. $\phi : cf \rightarrow cf'$. It takes a code fragment cf and changes it to a corresponding code cf'. There can be several such changes and we denote them by $\phi_1, \phi_2, \ldots, \phi_n$.

Consistent Change: All code fragments of a clone class in V_n appear in V_{n+1} after undergoing the same change.

$CC_n.cf_i = CC_{n+1}.cf_j' \wedge Num(CC_n) = Num(CC_{n+1})$ *where* $\phi(cf) = cf'$ represents change on code fragments.

Inconsistent Change: At least one code fragment from V_n changed differently than others in V_{n+1}. Different code fragments can undergo different types of changes.

$CC_n.cf_i = CC_{n+1}.cf_j'$ *where* $\phi_1(cf) = cf' \wedge CC_n.cf_i = CC_{n+1}.cf_j''$ *where* $\phi_2(cf) = cf''$ and so on.

2) EVOLUTION PATTERNS FOR STRUCTURAL CLONES

To study evolution in structural clones we need to define how structural clone classes change between the consecutive versions. For structural clones, clone evolution patterns need to be redefined because structural clones are structures (groups) of simple clones. Figure 6 shows some structural clone evolution patterns. Changes in a structural clone class can occur at the level of structures i.e. addition of a whole structure, and at the level of simple clones i.e. addition of new simple clones in existing structures. To study structural clone evolution systematically, we formally define their evolution patterns based on the evolution patterns of simple clones. SCC_n represents a structural clone class in V_n, and SCC_{n+1} represents its counterpart in V_{n+1} such that there exist a cloning relation between SCC_n and SCC_{n+1}. $SCC(V_n)$ represents set of all clones classes in V_n and $SCC(V_{n+1})$ represents set of all clone classes in V_{n+1}.

Same: None of the graphs of a structural clone class in V_{n+1} was changed from V_n. For all $CG_i \in SCC_n,\ 1 \leq i \leq Num(SCC_n.CG) \wedge$ for all $CG_j \in SCC_{n+1},\ 1 \leq j \leq Num(SCC_{n+1}.CG)$

$$SCC_n = SCC_{n+1} \text{ iff}$$
$$SCC_n.CG_i = SCC_{n+1}.CG_j$$

AND

$SCC_n.CG_i.cf_{k,i} = SCC_{n+1}.CG_j.cf_{k,j}$ where k represents the number of code fragments in each graph.

Addition of Graph: In V_{n+1}, at least one new graph is added in SCC

$SCC_{n+1} = SCC_n \cup \{CG_j\}$ where CG_j represents a new structural instance $\wedge\ p \geq j \geq 1$ where p is any finite number.

Addition of Node: In V_{n+1}, at least one new clone class is added in existing graphs of SCC.

$SCC_{n+1}.CG_i = SCC_n.CG_i \cup \{SCC_n.CG_i.cf_{l,i}\}$ *where* $cf_{l,i}$ represent new code fragments $\wedge\ p \geq l \geq 1$

Subtraction of Graph: In V_{n+1}, at least one new Graph is subtracted in SCC_n

$SCC_{n+1} = SCC_n - CG_j$ *where* CG_j represents a structural instance in $SCC_n \wedge p \geq j \geq 1$

Subtraction of Node: In V_{n+1}, at least one new clone fragment is subtracted in existing graphs of SCC_n.

$SCC_{n+1}.CG_j = SCC_n.CG_j - SCC_n.CG_j.cf_{l,j}$ *where* $cf_{l,j}$ represent code fragments in $SCC_n \wedge p \geq l \geq 1$

New: A new structural clone classSCC_{n+1} is introduced in V_{n+1}.

$$SCC(V_{n+1}) = SCC(V_n) \cup \{SCC_{n+1}\}$$

Removed: A structural clone class from V_n does not appear in V_{n+1}.

$$SCC(V_{n+1}) = SSC(V_n) - \{SCC_n\}$$

Consistent Change: All graphs of a structural clone class in V_n appear in V_{n+1} after undergoing the same change.

$SCC_n.CG_i = SCC_{n+1}.CG_j' \wedge Num(SCC_n) = Num(SCC_{n+1})$ *where* $\phi(CG) = CG'$ represents change on structural instances.

Inconsistent Change: At least one code fragment in any one graph of a structural clone in V_n changed differently than others in V_{n+1}.

$SCC_n.CG_i = SCC_{n+1}.CG_j'$ *where* $\phi_1(CG) = CG' \wedge SCC_n.CG_i = SCC_{n+1}.CG_j''$ *where* $\phi_2(CG) = CG''$and so on.

B. CLONE GENEALOGIES

To study clone evolution, a clone class in one version V_n is mapped to its counterpart in subsequent versions. A clone class can go through one or more change patterns (discussed in Section 4.1.2) between two versions. Clone lineage is a sequence of changes on a clone class in versions. Clone lineages ($CLIN$) represent the clone classes in versions i.e. CC_1 is a clone class in V_1 and CCn is the clone class in V_n.

$CLIN = \{CC_1, CC_2, CC_3, \ldots, CC_n\}$ where every $CC_i,\ 1 \leq i \leq n$, represents different versions of a clone class.

Clone genealogy is comprised of one or more clone lineages of a clone class [12]. Clone genealogy is the set of clone lineages belonging to same origin (i.e. clone class).

$CGEN = \{CLIN_1, CLIN_2, CLIN_3, ..., CLIN_m\}$ where every $CLIN_j$ starts from the same CC, $1 \leq i \leq m$. Clone genealogy helps in studying clone evolution semantically and structurally [12]. Clone genealogies represent the changes (or evolution patterns) in clone classes in various versions of a software system. Clone classes of a version are mapped to their counterpart in next versions and changes are observed between them. Fig. 3 shows a clone genealogy having a structural clone class containing two structural instances i.e. $\{CG_1, CG_2\}$ in V_n, which remain the same in V_{n+1}. In V_{n+2}, a new structural instance (CG_3) is added into it. In V_{n+3}, evolution pattern is also an 'Add pattern' but this time it is addition of instances of simple clone class $\{CC_1, CC_2, CC_3\}$ in existing structures. In V_{n+4}, a structural instance is subtracted from the class.

Clone genealogies can be categorized according to evolution patterns such as consistent/inconsistent genealogies or according to its life time such as dead genealogy and alive genealogy. Clone genealogies that remain in the system till the last release are called alive genealogies. Clone genealogies that disappear during software evolution are called dead genealogies.

C. MAINTENANCE IMPLICATIONS OF STUDYING STRUCTURAL CLONE EVOLUTION

Various studies performed on software clones suggest that clones require proper management. Detection of code clones in a software provides basic information (number of clone instances, length of cloned fragments) for the developers so that developers should be aware of the cloning in the software. To perform any maintenance task on a clone class, developers should know certain characteristics of a clone class e.g. program dependencies, history of that clone class. Study of clone evolution tells about evolutionary characteristics of clones which help developers in taking decisions during software evolution.

For efficient software maintenance, understanding of the system is required to deal with ripple effects of changes. If a developer is given a change task e.g. fixing a bug in cloned entities, a lot of information is required to perform this task. A developer needs to understand the change dependencies to analyze the impact of fixing a bug on other parts of the software. Simple clone class contains only a set of source code fragments whereas structural clones present a broader picture of cloning i.e. all the simple clone classes in those entities. This view of cloning will help developers to analyze change impact as compared to analyzing only source code fragment. For example, in Method Clone Classes, if a developer knows all the simple clone classes in those cloned methods, there are more chances that all parts that can be impacted will be detected. Evolution study of structural clones may improve change impact analysis because it gives historical perspective of the changing behavior of clone classes i.e. clone genealogy of an MCC reveals whether a clone class is changing frequently or infrequently, consistently or inconsistently in versions.

Another scenario is when addition of new feature/ functionality is required in a software. A developer will search if similar features exist in the software. For this type of change, developers need to understand the system at higher level. For example, if a new feature requires defining some new files/classes, then a developer needs to know which methods/functions need to be declared for these classes, which interfaces are required to be implemented, what would be their inheritance structure. Structural clones like FCC are

FIGURE 7. Proposed approach for clone genealogy analysis and extraction from software version repositories.

helpful in this type of analysis because these clones contain multiple cloning at method level as well as they contain same inheritance structures and same interface implementations. Structural clones of collaborating files/classes are also helpful in this case because they reveal cloning at higher level which gives information about clone files as well as execution flow among the clone entities. To take advantage of existing structural clones for reuse, it is necessary that the clones should be stable software entities (e.g. cloned files/classes). Evolutionary characteristics of clones such as clone lifetime help developers in estimating the stability of clone classes e.g. long lived clones are more reliable for reuse.

To improve the design of software systems, various refactoring or restructuring techniques are applied during software lifetime. Developers need to know which parts of the system require such improvements. Study of structural clone evolution helps in identifying those parts and taking appropriate decisions. For example, there are some clone classes that exist in a software for a long time, then developer may not be interested in refactoring them for removal or may give a low priority to their refactoring. Moreover, clone evolution study reveals interesting characteristics of clones e.g. clones that are regularly and consistently updated by developers but remain in the system for many versions. This information helps developers in tracking/managing them in future.

V. STUDY DESIGN

In this section, we describe the detailed setup for our study. Figure 7 shows the process of clone genealogy extraction from version repositories. In the first step, we selected subject systems from GIT repository [32]. To select subject systems, the first criterion was the availability of multiple versions, so that an evolutionary study can be performed on them. Secondly, they should be well known systems and should have been studied previously for clone research. For our experiments, we selected five Java systems i.e. JabRef, Xerces_J, Guava, JFreeChart and JHotDraw.

JabRef is a reference management system that provides a graphical interface to BibTeX and BibLaTeX style used by the LaTeX typesetting system [33]. Xerces_J is Apache's collection of software libraries for the purpose of parsing, validating and manipulating XML schema. Xerces_J uses various standard APIs for XML parsing [34]. Guava is basically a set of Java libraries commonly used by Google developers and suit of core utility functions which provides features like functional programming, graphs and caching [35]. JFreeChart is an open source Java chart library that provides user to make high quality charts for their applications [36]. JHotDraw is a Java framework for technical and structured drawing editors. It is very well known for software design patterns [32]. We selected some of the latest versions of these systems. Statistics of these software systems are given in Table 3.

TABLE 3. Studied software systems showing release information and size of systems.

Subject Systems	# of releases	Start date	End date	Starting release	# of methods
Jabref	13	Feb, 2015	Aug, 2016	2.1	4678
Xerces_J	14	Jan, 2003	Nov, 2010	2.3	5897
Guava	13	Oct, 2011	Dec, 2015	10	8998
JFreeChart	13	Dec, 2006	Nov, 2017	1.0.8	8526
JHotDraw	11	Feb, 2001	Jan, 2011	5.2	6344

A. CLONE DETECTION

To study clones in versions, we need to detect code clones for each version of software. For this purpose, we used the tool CloneMiner [8]. CloneMiner is a Java plug-in and detects simple clones and structural clones of a system. In our experiments, we set the minimum clone threshold of 30 tokens for simple clone detection because in literature it is shown that too small threshold value of tokens detects a large number of clones which may not be meaningful for maintenance and too large threshold value of tokens will detect a small number of clones and some important clones may be missed so token size of 30 token is more suitable for token based

TABLE 4. Statistics of simple clones and structural clones in subject systems.

Software Systems	Simple Clones			Structural Clones				
	Total # of clone classes	# of clone classes in versions (min - max)	Avg. # of instances per clone class (min - max) / standard deviation (min - max)	Total # of clone classes	# of clone classes in versions (min - max)	Avg. # of instances per clone class (min - max) / standard deviation (min - max)	Avg. # of simple clones per clone class (min - max)	# of simple clones contained by largest structural clone
Jabref	1414	95 - 156	2.1 - 3/ 0.5 - 4	106	6 - 13	2 - 2.1 / 0 - 0.6	5 - 11	44
Xerces_J	2695	161 - 264	2.3 - 2.5 / 1 - 1.5	377	21 - 35	2.1 - 2.5 / 0.2 - 1	6 - 9	33
Guava	2187	162 - 195	2.8 / 1.6	230	14 - 23	3 - 4 / 2.9 - 4.1	14 - 19	81
JFreeChart	2994	225 - 267	3.8 / 5.5	827	53 - 84	2.4 - 3 / 1.5 - 2.8	8 - 10	60
JHotDraw	1660	22 - 274	2.3 / 1.4 - 1.8	335	1 - 57	2.5 / 0.7	5 - 7	22

clone detection tools [8], [11]. No other customization were used for Clone Miner. CloneMiner detects a variety of structural clones discussed in Section 3.2. To extract these clones from CloneMiner tool, we used Navicat Premium tool [37] because CloneMiner visualizes clones in a GUI. Using Navicat, we accessed the database of CloneMiner and store the required clones in Excel files. CloneMiner detect structural clones at different levels such as method level (MCC), file level (FCC) and directory level (DCC). We selected FCC [8] to study evolution of structural clones because FCC represent structural clones of reasonable size and are more interesting for a maintainer as compared to MCC and DCC. MCC are the most basic type of structural clones and are smaller in size and less significant as compare to FCC. DCC represent much bigger clones and in some software systems, number of DCC may be very small. As we discussed in Section 3.2, FCC represent recurring patterns of Method Clone Structures (MCS) in files that fulfill a defined threshold of MCS. In this study, we selected all FCCs that contain at least one pair of MCC, for our analysis. $MCC_1 = \{mc_{11}, mc_{12}, mc_{13}\}$ and $MCC_2 = \{mc_{21}, mc_{22}, mc_{23}\}$, MCS will be as follows: $MCS_1 = (mc_{11}, mc_{21})$, $MCS_2 = (mc_{12}, mc_{22})$, $MCS_3 = (mc_{13}, mc_{23})$ where $FCC = \{MCS_1, MCS_2, MCS_3\}$ and its general notation will be $FCC = \{G_1, G_2, G_3\}$.

B. CHANGES EXTRACTION

Changes occurring in source code are stored in software version repositories such as GIT repository. We developed a web crawler in Java that extracts co-change information from the source repository. To analyze the evolution of clones, we extract the changes at release level because it is shown in previous studies that at revision level, users perform cloning on experimental basis so most of the clones at revision level are temporary clones [12].

C. CLONE GENEALOGY EXTRACTION

To study clone evolution, we need to track clone classes across various versions of the software. For this purpose, we developed an application in C#. Our application takes information of clone classes for various versions and co-change information which contains source code entities that are changed in the same change commit.

In a clone genealogy a clone class of a particular version is connected with all corresponding clone classes in the next version. To connect the clone classes of consecutive versions, we implemented the structural clone evolution patterns discussed in Section III. Based on these mappings, structural clone genealogies are extracted from multiple versions of software.

Various techniques have been used in the literature to map clone classes of different versions to each other [12], [13]. We used an approach based on CRD (Clone Region Description) [38]. CRD approach has been used in studying clone evolution of simple clones. In this approach, instead of source code, region of code fragment is mapped. Identifying the region of a simple clone at fragment level is difficult. But in our case, as we considered method level simple clones (detected by CloneMiner), identifying the region of clones is straightforward. CloneMiner reports the clone class as a group of methods, their file names and full path of the file. CRD of a clone class can be represented as 'method a( ) of file f in package p'. Where there is any ambiguity such as for one clone class in a particular version there are two clone classes in a subsequent version that are equally likely, we selected (included) both of them in a genealogy. If file of a particular clone instance is removed or renamed, location of that clone instance will be lost but as we applied the best match strategy (all clone instances will be matched with the instances of clone class in next version), a renamed or moved file will not result in the pattern of dead clones rather it will be considered as a shift pattern where one clone instance is removed and another is added.

VI. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we discuss the results of structural clone evolution. We first analyze the clone statistics of five Java software systems. Table 4 shows the number of clone classes, average number of clone instances and their standard deviation in a clone class for simple clones and structural clones of five Java systems. These averages are calculated in different versions of these systems. Table 4 shows the averages in min - max format which represent minimum and maximum averages in different versions e.g. 95 - 156 represents that the minimum average number of simple clone classes is 95 and the maximum average number of simple clone classes is 156 in studied versions of Jabref.

TABLE 5. Characteristics of clone genealogies for simple clones and structural clones.

Software Systems	Simple Clones							Structural Clones						
	Total Gen	Size of Gen		%age of Gen		Lifetime of Gen		Total Gen	Size of Gen		%age of Gen		Lifetime of Gen	
		Alive	Dead	Alive	Dead	Alive	Dead		Alive	Dead	Alive	Dead	Alive	Dead
JabRef	339	3	2.8	46	54	5.4	3.6	31	10.4	4.9	51.7	48.4	5.2	3.3
Xerces_J	382	2.4	2	69.2	30.9	10	4.5	48	7.8	6.3	73	27.1	9.5	5.1
Guava	380	2.9	2.4	49.3	50.8	9.5	3	35	24.4	12.9	48.6	51.5	10.4	3.5
JFreeChart	359	4.1	2.7	72.8	27.3	10	6.3	95	12.5	7.9	59	41.1	10.4	4.7
JHotDraw	413	2.4	3	66.4	33.7	4.7	2.5	85	6.9	6.2	60	40	4.8	2.1
Average	374.6	2.96	2.58	60.6	39.3	7.92	3.9	58.8	12.4	7.64	58.4	41.5	7.2	4.6

Average number of instances in simple clones and structural clones remains almost same in different versions. In Jabref, JFreeChart and JHotDraw, standard deviation of structural clones is much less than simple clones. For example, in JFreeChart, standard deviation of structural clones is 1.5 - 2.8 which is less than simple clones (5.5 in all versions). This indicates that structural clones classes are more stable than simple clones in terms of clone class size.

To perform any study on structural clones, it is very important to know about the *simple clone coverage* (i.e. how many simple clones are a part of structural clones) by structural clones in the studied systems. In the literature, it is reported that 54% of simple clones are represented in structural clones [11]. In our subject systems, 50%-60% simple clones are present in structural clones. This indicates that a large number of simple clones exist in some structural form i.e. recurring patterns of simple clones in files/directories. Presence of such a large number of simple clones in structural clones emphasizes on the need of structural clones to be studied for better software maintenance and evolution.

We also analyzed the *size of structural clone classes* i.e. how many simple clones are contained in a structural clone class. Table 4 shows the number of simple clones in the largest structural clone class for five software systems e.g. in Guava, largest structural clone class contains 81 simple clones. Average number of simple clone instances in structural clone classes is also shown in Table 4. Average size of structural clone classes varies for different subject systems such as the minimum average size is 5 simple clones for Jabref and JHotDraw and maximum average size is 19 for Guava.

RQ1: What are the characteristics of structural clones compared to simple clones in terms of clone lifetime?

Motivation: Lifetime of clones is a very important measure for identifying the relevance of clone types for clone management tasks. Clones that remain in the software for a very long time require different management as compared to those which disappear very quickly [22]. In the case of structural clones, long-lived clones may indicate intentional clones i.e. the clones are a deliberate outcome of a certain design decision and developers are aware of their presence in the software.

Results: Table 5 shows the *number and size of clone genealogies* in simple clones and structural clones for the test systems. Clones genealogies reduce the unrelated clone classes (shown in Table 4) into clone genealogies e.g. in JabRef, 1414 simple clone classes reduce to 339 clone genealogies and 106 structural clone classes reduce to 31 clone genealogies. Average clone size of genealogies in JFreeChart and Guava is greater than other systems for structural clones whereas it is greater in Jabref and JFreeChart for simple clones. Size of alive clone genealogies is more than dead clone genealogies for all subject systems. This indicates that smaller clone classes are more vulnerable to changes i.e. they disappear during software evolution whereas bigger clone classes sustain in the system for a long time.

Table 5 also shows the *lifetime* of alive genealogies and dead genealogies in simple and structural clones. In case of *alive genealogies*, there is very little difference (less than one version) between the lifetime of simple clones and structural clones in all subject systems. Lifetime of alive genealogies is substantially greater than lifetime of dead genealogies in both types of clones for all studied software systems. This finding is similar to [13], [22] in which clone genealogies of simple clones are analyzed. In our test systems, some alive genealogies remain in the system throughout all the studied versions. Life of *dead genealogies* is less than five versions in both structural clones and simple clones. It means that dead clone genealogies represent short lived (unsustainable) clones in the software and are removed from the systems earlier.

Number/percentage of alive genealogies is greater than percentage of dead genealogies in Xerces_J, JFreeChart and JHotDraw in both simple and structural clones whereas in other two systems, these are almost same for both types of clones. However, in most cases, percentage of dead genealogies of structural clones is greater than that of simple clones which indicates that structural clones are removed/refactored more often than simple clones in the studied systems.

Summary: Lifetime analysis of clone genealogies shows that there is no major difference in lifetime of simple clones and structural clones. However, some common patterns are observed for clone lifetime such as alive clone genealogies are generally greater in number, bigger in size and more sustainable than dead clone genealogies. It indicates that developers are aware about presence of alive genealogies in the system and are managing them regularly during software evolution. This information can help developers in software reuse such that developers can reuse long lived genealogies

with more assurity because these clones have been managed by developers for many versions. Moreover, for structural clones, alive clone genealogies represent persistent design information which also helps in understanding system design decisions. Lifetime of dead genealogies indicates that dead clone genealogies represent unsustainable clones which are removed from the system earlier. Moreover, dead clone genealogies have a smaller clone class size which indicates that smaller clones are more vulnerable to changes and disappear earlier in software lifetime. This information is also helpful for developers in prioritizing their work i.e. a developer should give priority (e.g. deciding about refactoring) to clones whose lifetime is less than five versions (average lifetime of dead genealogies in all subject systems).

RQ2: How consistently and frequently structural clones change as compared to simple clones?

Motivation: Frequency of change indicates the relation of clones to cost of maintenance i.e. a change in one clone instance may require changing other instances [22], [39]. Developers are more interested in frequently changing clones because any inconsistent change in them may increase cost of maintenance substantially.

Approach: To analyze the frequency of (in)consistent changes in simple clones and structural clones, we observed the changes at level of clone class because the purpose of our analysis is to investigate whether developers maintain clone instances of a clone class simultaneously or not. In the case of structural clones, a clone instance does not represent a code fragment but a group of code fragments (structural instance). We observed the changes in structural clones in two ways: a change in number of structural instances, and change in size of structural instances (simple clone instances added/deleted in structures). To analyze the consistent and inconsistent changes in structural clones, we also follow the same approach. For structural clones, a consistent change means that all simple clone instances in each structural instance of a structural clone class are changed in same change commit and remain clones of each other in the next version, thus preserving co-change similarity. An inconsistent change means that some simple clone instances in any structural instance of a structural clone class are changed in same change commit but some remain unchanged or they are changed in later commits or versions.

Results: Our results show that changes in structural clones are less frequent and more inconsistent than simple clones as shown in Figure 8. Low frequency of changes in structural clones indicates that structural clones incur less cost of maintenance than simple clones. As the size of structural clones is greater than simple clones, a high frequency means a considerable increase in maintenance cost. Low change frequency is a positive indicator with respect to maintenance cost.

The analysis of inconsistent changes shows that there are more inconsistent changes than consistent ones in simple clones. In structural clones, all the changes are inconsistent. The reason may be the larger size of structural clone classes

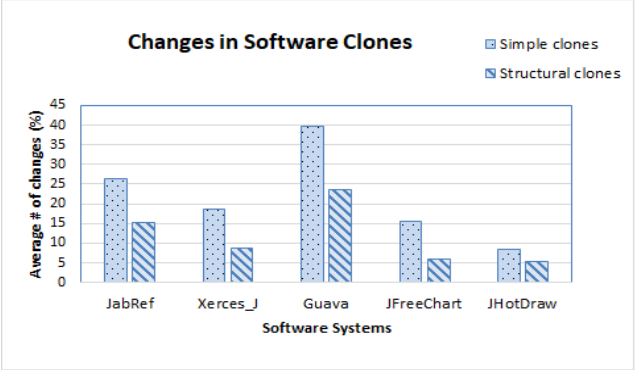

FIGURE 8. Changes in simple clones and structural clones.

i.e. in a class of ten clones (average coverage of simple clones in structural clone classes in our studied systems), there is less chance that all of them are changed in the same change commit. In previous clone evolution studies, it is observed that small sized clones are changed consistently than larger clones. For example, in a recent clone evolution study on micro clones revealed that consistent changes in micro clones are significantly greater than regular clones (as discussed in Section 2 also) [25]. Another observation is reported in [13], [22] that Type 2 and Type 3 clones are changed more inconsistently than Type1 clones. It is discussed in literature [40], inconsistent changes in code clones can negatively impact software quality because an inconsistent change may introduce bugs in software and it may incur additional cost of maintenance.

Analysis of inconsistently changing clone classes shows that in most cases, proportion of changed clone instances is less than non-changing clone instances in a structural clone class in all the five Java systems. For example, if there are eight instances in a structural clone class, only one or two instances are changed whereas others remain unchanged. In JFreeChart, all changes in structural clones are inconsistent (19% of total structural clone classes) but the ratio of clone instances of these classes is less than 25% of total instances.

There may be different reasons of inconsistent changes in structural clones. Structural clones are groups of different simple clone classes which are grouped together based on cloning patterns in software. As structural clones consist of different simple clone classes which have different code implementations, a change in one clone class may not be required in other clone classes. Another reason of inconsistent changes in structural clones may be the lack of tool support for structural clones. As there is no industrial level tool available for tracking structural clones in software, developers may not be aware of the presence of structural clones in a system and thus may make inconsistent changes. In case of structural clones, inconsistent changes may not be harmful for software quality (i.e. may not introduce bugs) but they may result in information loss of software design similarity. As structural clones help in understanding software design, loss of a structural instance may be harmful in

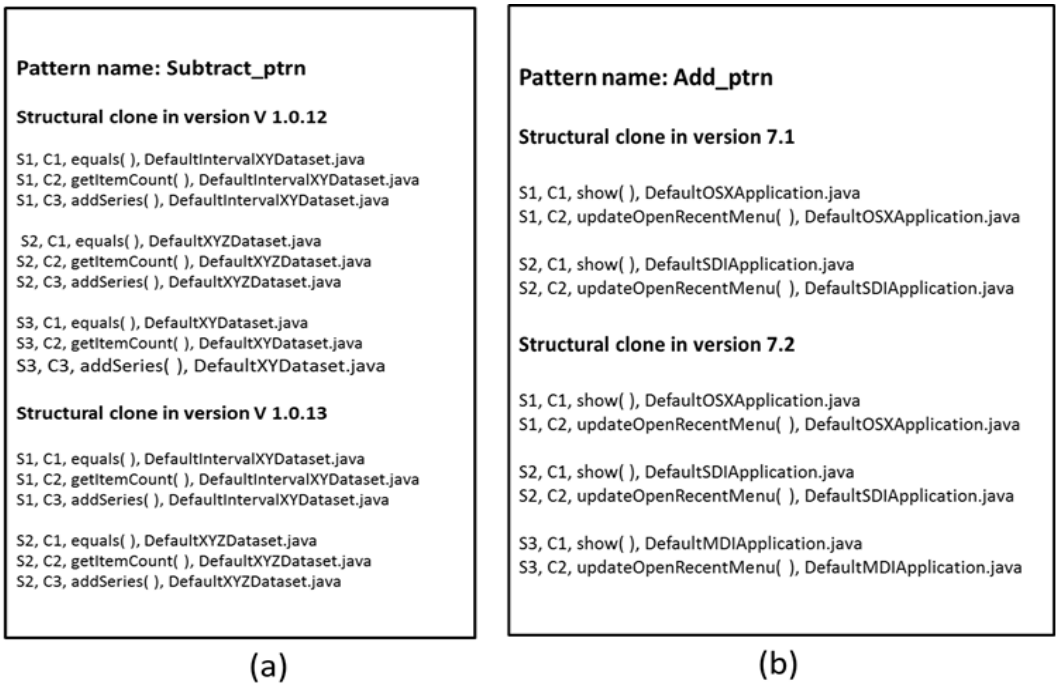

FIGURE 9. Examples of evolution pattern (a) Subtract pattern and (b) Add pattern.

software understanding. A tool for structural clone management can help in making more conscious changes in structural clones.

Summary: Our results show that changes in both simple clones and structural clones are mostly inconsistent. There may be different reasons of inconsistent changes. For example, developers are not aware about the presence of clones in the system and they make the change only on the clone instance that is involved in the current maintenance task and leave others. These results help in deciding about managing clones such as consistently changing clones are more suitable candidates for refactoring whereas inconsistently changing clones need to be tracked to assist developers in making further required changes in clones during evolution. Some studies show that inconsistent changes may not necessarily introduce bugs or impact the software negatively [30]. In some cases, clones need to be evolve differently to meet the new requirements. To understand the reason of inconsistent changes in structural clones, we investigate them manually in our next research question (RQ3).

RQ3: How structural clone classes change between the versions in evolving software?

Motivation: In depth analysis of changes in structural clones in versions tells how structural clones are actually changed during software evolution. An analysis of reasons of changes in structural clones will help in their better management.

Approach: To answer this question, we analyze how a clone class changes between consecutive versions e.g. what type of evolution pattern is frequently occurring in a clone class during evolution. We explore the actual source code changes that made the evolution pattern occur. In the following we will discuss some examples to understand structural clone evolution.

Examples: In case of structural clones, a clone instance represents a group of simple clone instances belonging to different simple clone classes. In Figure 9(a), a structural class contains three structural clone instances comprised of instances of different simple clone classes. To illustrate examples of evolution patterns, we define the following parameters for a structural clone instance.

Struct_ins_id, clone_ins_id, method_name, file_name, where struct_ins_id denotes a structural instance, clone_ins_id denotes simple clone instance which is part of the structural clone instance, method_name, denotes name of the method where simple clone instance resides, file_name denotes name of the file where the method is declared.

Figure 9(a) illustrates an example of subtract pattern. There are three structural instances S1, S2, S3 in Version 1.0.12 of JFreeChart. In these instances three similar clones C1,C2,C3 are repeatedly occurring in methods of three files (DefaultIntervalXYDataset.java, DefaultXYDataset.java, and DefaultXYZDataset.java.). According to parameters, first instance of this structural clone class can be defined as follows: S1, C1, equals( ), DefaultIntervalXYDataset.java

In next version, S3 is subtracted from the structural clone class. To understand the reason of this evolution pattern (subtraction of a structure), we analyzed the source code of both versions. Analysis of a subtract pattern shows that clones of

TABLE 6. Clone evolution patterns for simple and structural clones in versions of subject systems.

Software Systems	Simple Clones					Structural Clones				
	Same	Add	Subtract	New	Shift	Same	Add	Subtract	New	Shift
JabRef	75%	3%	2%	18%	4%	82%	5%	6%	20%	3%
Xerces_J	90%	2%	1%	9%	2%	85%	3%	3%	8%	3%
Guava	85%	3%	2%	10%	3%	76%	6%	5%	10%	5%
JFreeChart	91%	2%	3%	5%	2%	64%	6%	8%	5%	5%
JHotDraw	58%	2%	2%	39%	2%	64%	8%	8%	31%	6%
Average	79.80%	2.40%	2%	16.20%	2.60%	74.20%	5.60%	6%	14.80%	4.40%

these files are similar in functionality but dimension points of charts are a little different in all the three files. In Version 1.0.13, more checks are applied in equals() method of DefaultXYZDataset.java and DefaultIntervalXYDataset.java but DefaultXYDataset.java remained unchanged. That's why it does not remain a part of structural clone in next version. An inconsistent change in one simple clone instance is the reason of this subtraction from the structural clone class.

Some other examples of inconsistent changes show that sometimes the reason of an inconsistent change is the correction of an error in implementation. For example, in another structural clone containing multiple instances of equals methods in different files, the equals() methods are inconsistently changed to remove some implementation anomalies. In [41], Rupakhet et al. also discussed the implementation anomalies of equals() method in Java applications. These examples indicate that with time, clones may become more correct and reliable to be reused.

Another example is of an Add pattern shown in Fig 8 (b). In this example, it can be seen how an add pattern changes the structural clone class. This pattern represents two types of changes on a structural clone class as discussed in Section 3 i.e. addition of a structural instance and addition of simple clones in existing structures. In the following we will discuss an example of both cases.

Figure 9(b) illustrates an example of Add pattern where a structural clone class consists of two structural instances S1, S2 in Version 7.1 of JHotDraw. In next version, S3 is added in the structural clone class. Source code analysis of both versions indicates that clones entities of newly added structure i.e. show( ) and updateOpenRecentMenu( ), methods of file DefaultMDIApplication.java already exist in the Version 7.1. It is similar to the methods of other two files according to functionality but different in implementation. To show drawings on panels it uses JFrames whereas other two methods are implemented through JInternalFrame. In next version, developer changed the implementation of show() method of DefaultMDIApplication.java and this method becomes part of the clone class. Analysis of other examples of change patterns reveal that two reasons are commonly observed for newly added structures. Methods already exist in previous versions but are not similar enough to become the part of clone class. Some code changes in the next version make addition of new clone pairs in the existing clone structures. This shows that with time, more simple clones classes become a part of existing structural clones in consecutive versions. In other cases, a new method is added in a file and it becomes a part of clone class because the same method(s) already exists in the other file(s). Addition of new method makes another simple clone class part of the existing structures.

Summary: Our analysis reveals that independent evolution of clone instances is the main reason of inconsistent changes in structural clone classes. In most cases, when new functionality is added, developers write code without following the software design, or standard implementations. But in next versions, code is refined or refactored which make changes in clones. This analysis helps developers in understanding evolution from the perspective of system design such as code that does not follow the design may need to be updated in future to remove the inconsistency. Further analysis of evolution patterns revealed some interesting situations where methods with similar functionality already exist in previous versions but their similarity with other clone methods is less than the defined similarity threshold set for clone detection tool. That is why they did not become the part of clone class in the current version. Source code changes in next version make it a part of existing structural clone class.

RQ4: How evolution of structural clones differs from the evolution of simple clones between the versions?

Motivation: Study of clone evolution patterns between the versions will help developers in understanding the kind of changes occurring in software clones during their evolution. Finding trends in the clone evolution patterns will help in software understanding and will be a useful guide in devising better clone management systems.

Approach: To investigate how structural clones change in versions, we analyzed the structural clone evolution patterns as defined in Section 3. Evolution patterns of simple clones are also analyzed for comparison with structural clones for all the studied versions.

Results: Table 6 shows the *percentage of evolution patterns* in simple and structural clones in different versions for five Java systems. Most frequent pattern is 'Same pattern' i.e. more than 50% clones remain same between the versions for all subject systems for both simple and structural clones. Second most frequent pattern is 'New pattern' which shows that new clone classes have been added in versions. Analysis of 'New pattern' in all studied versions of five Java systems revealed that new clones have been added consistently

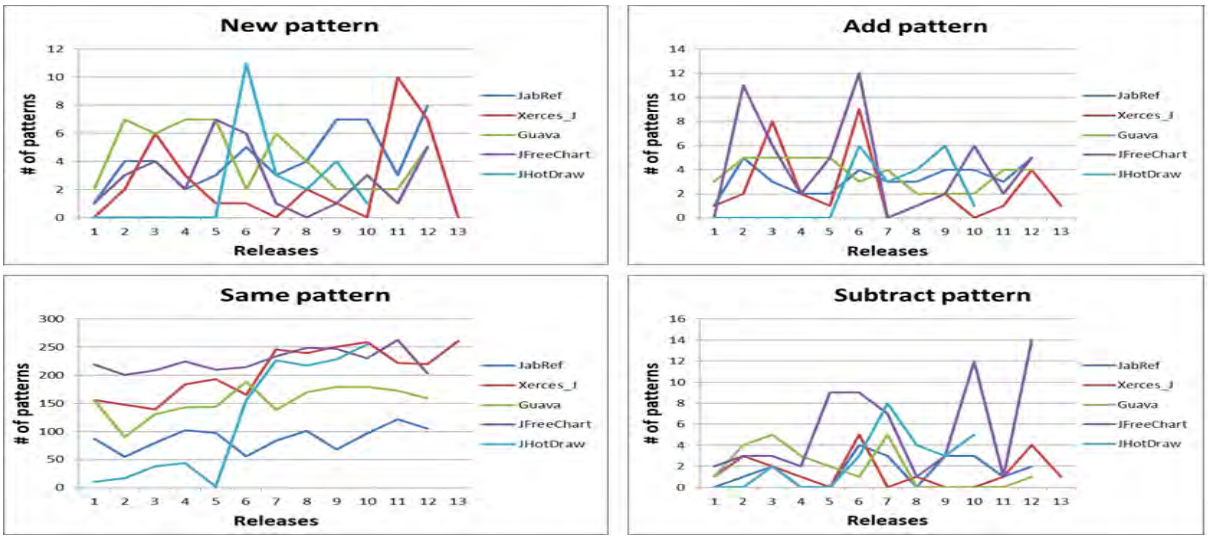

FIGURE 10. Evolution patterns of simple and structural clones in various versions of five Java systems.

between the versions. Frequency of 'New pattern' is almost same in simple and structural clones. In JHotDraw, 'New pattern' is more frequent (39% in simple clones and 31% in structural clones) than other four systems (maximum 20% in JabRef). Analysis of source code and change data of JHotDraw showed that JHotDraw 7.1, is a major departure from previous versions of JHotDraw because only the basic architecture remains same whereas the API and almost every part of the implementation have been reworked to take advantage of the Java SE 6 platform [32]. This is the reason of significant increase in number of new clones in JHotDraw. Other evolution patterns are Add, subtract, shift and Dead patterns. Frequencies of these patterns range from 1% to 10% in both simple clones and structural clones which represent these are relatively uncommon patterns. These patterns are more frequent in structural clones than simple clones. These patterns are important to analyze in case of structural clones because they represent change in system design e.g. 'Add pattern' shows that more code entities are becoming part of existing structural clones. This may indicate that an existing design is becoming more stable.

We also analyzed *evolution patterns in versions*. Figure 10 shows evolution patterns in different versions of five Java systems for simple clones. In analysis of evolution patterns between the versions, it is observed that all evolution patterns are following a zigzag pattern. It means that number of evolution patterns are high in one or two versions and then they significantly decreased in next few versions (again one or two mostly). After that they again increased in next version. For example, in JFreeChart, number of 'add patterns' are high (11) in 2nd version, and then in next three versions, they remain 2 to 6 and then in the 6th version number of 'add pattern' are high (12) again as shown in Fig. 9. similar situation can be observed in Xerces_J also. Changes in clone classes (e.g. 10 add patterns are observed) in few versions, and then in next few versions, the number of changes is very less.

In Xerces_J, number of 'add patterns', subtract and shift patterns are high in 3rd version (version 2.6.0) but in in next two versions they remain 1 to 3 and then again high in next version (version 2.7.0). One reason is that these are the main releases of Xerces_J. Major changes such as compliance of XercesJ with most recent APIs are carried out in major versions. For example, in V2.7.0, a complete implementation of the parser related portions of JAXP 1.3 is incorporated and compliance of Xerces with other APIs e.g. SAX 2.0.2 is provided. A new package 'org.apache.xerces.xs.datatypes' has been added to Xerces' XML Schema API that provides a full schema data type to object mapping.

We also analyzed the *clone evolution patterns system wise* as shown in Figure 11. All systems show similar trends e.g. number of 'same pattern' and new pattern are generally greater in all the systems. JFreeChart and JHotDraw show a different behavior in some cases. In JFreeChart, number of 'New pattern' is less than other systems. Analysis of release history of JFreeChart shows that these are regular releases. New features have been added in the releases but new clone classes are less. JHotDraw shows a different behavior than other systems because 'New pattern' and 'Same pattern' show a high peak in version 7.1 and 7.2 (version 4 and 5 according to Figure) respectively. As discussed above, changes are applied in all parts of JHotDraw 7.1 to take advantage of the Java SE 6 platform. Out of 44 clone classes in version 6.0, only two simple clone classes remain same whereas for structural clones, no clone class remains same. In next version, there is significant increase in number of clones for other patterns e.g. 'same pattern' and 'add pattern' are significantly increased. A number of new clones are introduced in 7.2 also which indicates that the impact of changes made in previous version has been continued in

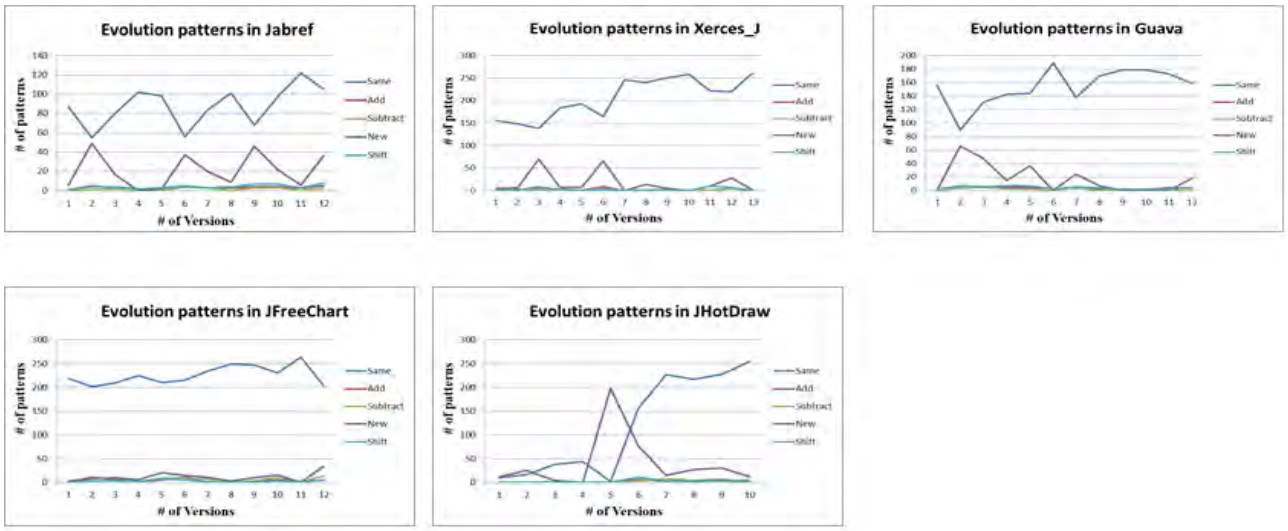

FIGURE 11. System wise trend of clone evolution patterns in versions of five Java systems.

next versions. In version 7.3, software seems getting stable as most clones remain same while a few 'add' and 'subtract pattern' is observed. This also verified the pattern that even after major changes are applied in software, software clones gets stable in a few releases.

Another different behavior is observed in JFreeChart, where number of 'New pattern' is less than other systems. There are no visible peaks in 'New pattern' as compared to other systems. Analysis of release history of JFreeChart shows that major changes occurred in some releases. New features have been added in the releases but new clone classes are less. We analyzed the 'New pattern' in JFreeChart and other systems to find the reasons of new clones being added in the system in subsequent versions. Usually new clone classes emerge in next version as a result of addition of similar functionality. For example, a method with particular functionality already exists in a version, a new method with similar functionality is added in next version, which introduces new clone classes. In JFreechart, all releases are regular releases with minor improvements and bug fixes. Addition of new functionality is less as compared to other systems.

Summary: Analysis of evolution patterns of simple clones and structural clones in versions reveals that all evolution patterns are following similar trends in all subject systems i.e. number of evolution patterns are greater in one or two versions and then they significantly decreased in next few versions and henceforth they again increased. In cases, where major changes occur in a version (e.g. in JHotDraw 7.1), number of evolution patterns increased significantly but after one or two releases number of evolution patterns decreases. This evolution trend shows that major releases make major changes in software clones but clones get stable in a few releases. Analysis of evolution patterns show that major changes in clones occur in major releases i.e. new clone classes are added, clone instances are added and deleted in existing clone classes. In major releases, new features have been implemented, existing implementations have been improved and major and minor bugs have been fixed. This analysis helps developers in understanding the evolution trends in clones such as with the addition of new features in major releases, software stability may deteriorate and to counter this, a good understanding of software cloning and an intelligent clone management is required.

VII. CONCLUSION

In this paper, we presented an empirical study on the evolution of structural clones in different versions of five Java systems. We presented a formal definition of structural clones and their evolution patterns. Based on these patterns we extracted structural clone genealogies to study evolution of structural clones. We also compared the evolutionary characteristics of structural clones with simple clones to know whether design level similarities evolve differently than textual similarities. Our analysis on five Java systems shows that understanding structural clone evolution is helpful for software maintenance tasks.

Study of software clones in versions gives quantitative facts that lead to qualitative inferences/implications about the evolutionary behavior of clones. These implications will be helpful for maintainer in clone management during software evolution. For example, lifetime analysis of clone genealogies in our studied systems showed that life of alive clone genealogies is greater than dead clone genealogies for both simple clones and structural clones. This implies that alive clone genealogies represent sustainable clones and developers are managing them regularly during evolution of the software systems. These findings will help developers in prioritizing their work for future software maintenance such as attending to long lived clones more thoroughly than short lived clones.

Inconsistent change in structural clones indicates that developers may not be aware of the presence of structural clones in the system. One reason may be that structural

clones represent design level similarities and managing them requires a high level understanding of the system. As there is less support of tools to understand high level cloning in the system, developers may not be able to manage structural clones properly. As structural clones evolve inconsistently, are bigger in size and encompass some design similarity in software, they need more intelligent clone management systems. We believe that our study on structural clone evolution will help in devising tools to support proper management of structural clones. In future, we will include more subject systems of other languages such as C++ to verify the generalizability of our findings regarding structural clone evolution.

REFERENCES

[1] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future," in *Proc. Softw. Evol. Week-IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng.*, Feb. 2014, pp. 18–33.
[2] S. Jarzabek and S. Li, "Unifying clones with a generative programming technique: A case study," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 18, no. 4, pp. 267–292, 2006.
[3] C. J. Kapser and M. W. Godfrey, "'Cloning considered harmful' considered harmful: Patterns of cloning in software," *Empirical Softw. Eng.*, vol. 13, no. 6, pp. 645–692, 2008.
[4] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School Comput.*, vol. 541, no. 115, pp. 64–68, 2007.
[5] S. Rahman and C. K. Roy, "A change-type based empirical study on the stability of cloned code," in *Proc. 14th Int. Work. Conf. Source Code Anal. Manipulation SCAM*, 2014, pp. 31–40.
[6] M. Zibran, "Management aspects of software clone detection and analysis," Ph.D. dissertation, Dept. Comput. Sci., Univ. Saskatchewan, Saskatoon, SK, Canada, 2014.
[7] N. Göde, "Clone removal: Fact or fiction?" in *Proc. 4th Int. Workshop Softw. Clones*, 2010, pp. 33–40.
[8] H. A. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 497–514, Jul. 2009.
[9] H. A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," *ACM Sigsoft Softw. Eng. Notes*, vol. 30, pp. 156–165, Sep. 2005.
[10] H. A. Basit and S. Jarzabek, "A case for structural clones," in *Proc. Int. Workshop Softw. Clones*, 2009, pp. 7–11.
[11] H. A. Basit, U. Ali, S. Haque, and S. Jarzabek, "Things structural clones tell that simple clones don't," in *Proc. 28th Int. Conf. Softw. Maintenance*, 2012, pp. 275–284.
[12] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 187–196, Sep. 2005.
[13] S. Bazrafshan, "Evolution of near-miss clones," in *Proc. 12th Int. Work. Conf. Source Code Anal. Manipulation*, 2012, pp. 74–83.
[14] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *Proc. 11th Eur. Conf. Softw. Maintenance Reeng.*, 2007, pp. 81–90.
[15] P. Jablonski and D. Hou, "CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE," in *Proc. OOPSLA Workshop Eclipse Technol. Exchange*, 2007, pp. 16–20.
[16] F. Jacob, D. Hou, and P. Jablonski, "Actively comparing clones inside the code editor," in *Proc. 4th Int. Workshop Softw. Clones*, 2010, pp. 9–16.
[17] D. Hou, F. Jacob, and P. Jablonski, "Exploring the design space of proactive tool support for copy-and-paste programming," in *Proc. Conf. Center Adv. Stud. Collaborative Res.*, 2009, pp. 188–202.
[18] D. Hou, P. Jablonski, and F. Jacob, "CnP: Towards an environment for the proactive management of copy-and-paste programming," in *Proc. 17th Int. Conf. Program Comprehension*, 2009, pp. 238–242.
[19] J. Kanwal, H. A. Basit, and O. Maqbool, "Structural clones: An evolution perspective," in *Proc. 12th Int. Workshop Softw. Clones*, 2018, pp. 9–15.
[20] R. Koschke, "Survey of research on software clones," in *Proc. Dagstuhl Seminar*. Wadern, Germany: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007, pp. 1–24.
[21] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone evolution: A systematic review," *J. Softw., Evol. Process*, vol. 25, no. 3, pp. 261–283, 2013.
[22] R. Saha, C. Roy, K. A. Schneider, and D. E. Perry, "Understanding the evolution of type-3 clones: An exploratory study," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 139–148.
[23] M. Mondal, C. K. Roy, and K. A. Schneider, "An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study," *Sci. Comput. Program.*, vol. 95, pp. 445–468, Dec. 2014.
[24] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug-proneness and late propagation tendency of code clones: A comparative study on different clone types," *J. Syst. Softw.*, vol. 144, pp. 41–59, Oct. 2018.
[25] M. Mondai, C. K. Roy, and K. A. Schneider, "Micro-clones in evolving software," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2018, pp. 50–60.
[26] F. Zhang, S.-C. Khoo, and X. Su, "Predicting change consistency in a clone group," *J. Syst. Softw.*, vol. 134, pp. 105–119, Dec. 2017.
[27] M. Mondal, S. Rahman, C. K. Roy, and K. A. Schneider, "Is cloned code really stable?" *Empirical Softw. Eng.*, vol. 23, no. 2, pp. 693–770, 2018.
[28] W. Qian, X. Peng, Z. Xing, S. Jarzabek, and W. Zhao, "Mining logical clones in software: Revealing high-level business and programming rules," in *Proc. 29th Int. Conf. Softw. Maintenance ICSM*, 2013, pp. 40–49.
[29] Y. Lin *et al.*, "Clonepedia: Summarizing code clones by common syntactic context for software maintenance," in *Proc. Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 341–350.
[30] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *Proc. 14th Work. Conf. Reverse Eng.*, 2007, pp. 170–178.
[31] C. K. Roy and J. R. Cordy, "Benchmarks for software clone detection: A ten-year retrospective," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, 2018, pp. 26–37.
[32] (2018). *JHotDraw*. [Online]. Available: http://www.jhotdraw.org//
[33] (2018). *JabRef*. [Online]. Available: http://www.jabref.org/
[34] (2018) *Xerces*. [Online]. Available: http://xerces.apache.org/
[35] (2018). *Guava*. [Online]. Available: https://github.com/google/guava
[36] (2018). *JFreeChart*. [Online]. Available: http://www.jfree.org/jfreechart/
[37] (2019). *Navicat*. [Online]. Available: https://www.navicat.com/en/store/navicat-premium/
[38] E. Duala-Ekoko and M. p. Robillard, "Tracking code clones in evolving software," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 158–167.
[39] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at release level," in *Proc. 16th Work. Conf. Reverse Eng.*, 2009, pp. 85–94.
[40] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Sci. Comput. Program.*, vol. 77, no. 6, pp. 760–776, 2012.
[41] C. Rupakheti and D. Hou, "An empirical study of the design and implementation of object equality in java," in *Proc. Conf. Center Adv. Stud. Collaborative Res., Meeting Minds*, 2008, Art. no. 9.

JAWERIA KANWAL received the M.Phil. degree in computer science from Quaid-i-Azam University, Islamabad, Pakistan, in 2011, where she is currently pursuing the Ph.D. degree in computer science.

She was a Research Fellow with the Graduate School of Information Science and Technology, Osaka University, Japan, in 2016. She was a Research Fellow with the Computer Science Department, Lahore University of Management Sciences, Pakistan, in 2017. She has published in well reputed software engineering conferences and journals. Her research interests include mining software repositories to study evolutionary aspects of software systems, clone analysis, machine learning, software maintenance and evolution, and software refactoring.

ONAIZA MAQBOOL received the Ph.D. degree in computer science from the Lahore University of Management Sciences, in 2006.

She was with the software industry for some years. She is currently an Associate Professor with the Department of Computer Science, Quaid-i-Azam University, Islamabad, Pakistan. She has published more than 30 papers in well-reputed journals and conferences. Her research interests include exploring machine learning techniques to solve software engineering problems and other research areas.

HAMID ABDUL BASIT received the Ph.D. degree from the National University of Singapore, where he held a postdoctoral position. He then taught at the Computer Science Department, Lahore University of Management Sciences, as a full time Faculty Member. He is currently supervising the development of a complete clone management tool suite that handles both simple and structural clones. He is also a Software Engineering Researcher and a Practitioner. He has published in top software engineering conferences and journals. His research interests include software maintenance, software reuse, and code recommendation systems.

MUDDASSAR AZAM SINDHU received the M.Sc. degree in computer science from Punjab University, Lahore, Pakistan, in 2004, and the Licentiate degree in engineering and the Ph.D. degree in computer science from the Royal Institute of Technology (KTH), Stockholm, Sweden, in 2011 and 2013, respectively.

He is currently an Assistant Professor of computer science with Quaid-i-Azam University, Islamabad, Pakistan. His research interests include automatic test case generation process from both formal and informal requirements of software, and grammatical inference of software systems along with developing techniques and tools for analyzing/testing of safety and security vulnerabilities in software.