The University of Saskatchewan
Department of Computer Science

Technical Report #2012-03

UNIVERSITY OF SASKATCHEWAN

The Road to Software Clone Management: A Survey

Minhaz F. Zibran Chanchal K. Roy
{minhaz.zibran, chanchal.roy}@usask.ca
University of Saskatchewan, Canada

February 13, 2012

Contents

List of Figures

List of Tables

1 Introduction and Motivation

Copying an existing code and pasting it in somewhere else followed by major or minor edits is a common practice that the developers adopt to increase productivity. Such a reuse mechanism typically results in duplicate or very similar code fragments residing in the code base. Those duplicate or near-duplicate code segments are commonly known as code clones. There are many reasons why the developers intentionally perform such code cloning. Obvious reasons include reuse of existing implementation without re-inventing the wheel. More comprehensive discussions on the reasons for code cloning can be found elsewhere [106, 161, 163]. Code clones may also appear in the code base without the awareness of the developers. Such unintentional/accidental clones may be introduced, for example, due to the use of certain design patterns, use of certain API's to accomplish similar programming tasks, code conventions imposed by the organization, and so on.

The reuse mechanism by code cloning offers some benefits. For instance, cloning of an existing code that is already known to be flawless, might save the developers from probable mistakes they might have made if they had to implement the same from the scratch. It also saves the time and effort in devising the logic and typing the corresponding textual code. Code cloning may also help in decoupling classes or components and facilitate independent evolution of similar feature implementations.

On the other end of the spectrum, code clones may also be detrimental in many cases. Obviously, redundant code may inflate the code base, and may increase resource requirements. This may be crucial for embedded systems and systems such as hand held devices, telecommunication switches, and small sensor systems. Moreover, cloning a code snippet that contains any unknown fault may result in propagation of that fault in all copies of the faulty fragment. From the maintenance perspective, a change in one code segment may necessitate consistent changes in all clones of that fragment. Any inconsistency may introduce bugs or vulnerabilities in the system. Fowler et al. [58] recognize code clones as a serious kind of code smell.

However, during the software development process, duplication cannot be avoided at times. For example, duplication may be enforced by the limitation of the programming language in facilitating with necessary mechanism to implement an efficient generic solution of a problem at hand. Code generators may also generate duplicated code, that the developers do not want to modify.

Previous research reports empirical evidences that a significant portion (generally 9%-17% [206]) of a typical software system consists of cloned code, and the proportion of code clones in the code base may be as low as 5% [163] and as high as even 50% [162]. Indeed, due to the negative impact of code clones in the maintenance effort, one might want to remove code clones by active refactoring, wherever feasible. However, in reality, aggressive refactoring of code clones appears not to be a very good idea [39], and not all clones are really removable through refactoring. Due to the dual

role of code clones in the development and maintenance of software systems, as well as the pragmatic difficulty in avoiding or removing those, researchers and practitioners have agreed that code clones should be detected and managed efficiently.

Since the emergence of software clones as a research area, significant contributions over years made the field grow and become quite a matured area of research. Nonetheless, over the entire course of software clone research there have been notably three surveys. Koschke [125], in 2007, presented a brief summary of the important findings about different aspects of software clones including cause-effect of cloning, clone avoidance, detection, and evolution. along with a set of open questions. In the same year, Roy and Cordy [163] also published another survey containing a thorough review on those same areas with specific focus on clone the detection tools and techniques. Recently, Pate et al. [159] published a systematic review on 30 studies on clone evolution only. In this paper, we present an extensive survey on code clone research with strong emphasis on clone management.

This paper is organized as follows. In Section 2, we present a systematic review on a repository of 262 papers on software clone research published over 19 years. The review draws "birds-eye" view on the overall contributions and growth along different dimensions of software clone research. The remaining of this survey is the outcome of careful investigation of literature beyond the said repository, and through analysis in the light of our experience. Section 3 starts with the necessary background including the definition and types of clones. Section 3.3 describes the different granularities, at which code clones can be addressed. Section 3.4 distinguishes intentional and accidental clones. In Section 3.5, we describe the different clone detection techniques along with their strengths and weaknesses. Section 3.6 addresses clone evolution with emphasis on clone change patterns based on the genealogy based evolution model.

Section 4 begins with the management of clones beyond detection. Section 4.1 characterizes different strategies for clone management. In Section 4.2, we briefly describe the design space for a clone management system. Section 4.3 starts discussion on the different clone management activities, including integrated clone detection (Section 4.4), clone documentation (Section 4.5), tracking (Section 4.6) and annotation (Section 4.7). Section 4.8 presents the techniques for clone removal or clone based reengineering. In Section 4.9, we describe the analyses for the identification of potential clones as candidates for refactoring/reengineering. The cost-benefit analysis and scheduling of clone refactoring is then presented in Section 4.10. A discussion on the verification of clone refactoring is accommodated in Section 4.11. Our understanding on the challenges for industrial adoption of clone management is presented in Section 5, and finally, Section 6 concludes the paper.

2 A Systematic Review on Clone Literature

There has been more than a decade of research in the field of software clones. To understand the growth and trends in the different dimensions of clone research we carried out a quantitative review on the related publications. Robert Tiras at the University of Alabama at Birmingham has been maintaining a repository [181] of scholarly articles that make significant contributions in the area. Until today, the corpus consists of 264 scholarly articles published between 1994 and 2012 in different refereed venues including Ph.D., M.Sc., and Diploma theses. The repository organizes the publications by categorizing them based on their contributions in four major sub-areas of clone research. The categories are as follows:

Analysis: This category contains publications that perform analysis on the various traits of software clones, their reasons, existence, effects in software systems, as well as investigation of clone reengineering opportunities and implications. A majority of such publications report findings from qualitative or quantitative empirical studies.

Detection: Publications in this category address techniques and tools for the detection of software clones.

Tool Evaluation: This category comprises the publications that contribute to the quantitative or qualitative evaluation of the techniques and tools for clone detection.

Management: Publications in this category address the issues, techniques and tools for the management of code clones beyond detection.

Other than the publications that fall in the aforementioned four categories, the corpus also contains three publications categorized as "Survey of Overall Research". After careful reading of those articles, we put them in the 'Analysis' category. Moreover, the repository classified the published theses based on simply whether they were outcomes of Ph.D., M.Sc., and Diploma programs. We also went through them, and based on their research focus, we classified them (Table 1) into the above mentioned four categories.

In our review, we include a total of 262 scholarly articles and theses published since 1994 until 2012. Since, it is now early 2012 and more contributions are expected to appear by the end of 2012, we deliberately excluded publications after 2011 for sanity. The repository also contains a list of tools, events, research groups, and links relevant to software clone research, which we excluded from the review. We collected the information by performing automated parsing (followed by manual verification) of the web (HTML) interface of the repository.

Table 1: Categories of theses on software clone research

	Title	Author	Year	Category
Ph.D. Thesis	Representation, Analysis, and Refactoring Techniques to Support Code Clone Maintenance	R. Tairas	2010	Management
	Scalable Detection of Similar Code: Techniques and Applications	L. Jiang	2009	Detection
	Toward an Understanding of Software Code Cloning as a Development Practice	C. Kapser	2009	Management
	Assessing the Effect of Source Code Characteristics on Changeability	A. Lozano	2009	Analysis
	Detection and Analysis of Near-Miss Software Clones	C. Roy	2009	Detection
	Code Clone Analysis Methods for Efficient Software Maintenance	Y. Higo	2006	Analysis
	Effective Clone Detection Without Language Barriers	M. Rieger	2005	Detection
	Automated Duplicated-Code Detection and Procedure Extraction	R. Komondoor	2003	Detection
M.Sc. Thesis	Improving Clone Detection for Models	P. Pfaehler	2009	Detection
	Clone Detection Using Dependence Analysis and Lexical Analysis	Y. Jia	2007	Detection
	Clone Detection Using Pictorial Similarity in Slice Traces	Y. Jafar	2007	Detection
	Visualizing and Understanding Code Duplication in Large Software Systems	Z. Jiang	2006	Analysis
Diploma Thesis	Incremental Clone Detection	N. Göde	2008	Detection
	CPC: An Eclipse Framework for Automated Clone Life Cycle Tracking and Update Anomaly Detection	V. Weckerle	2008	Management
	Semi Automatic Removal of Duplicated Code	Y. Liu	2004	Management
	Automated Detection Of Code Duplication Clusters	R. Wettel	2004	Detection
	A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems	G. Koni-N'Sapu	2001	Management

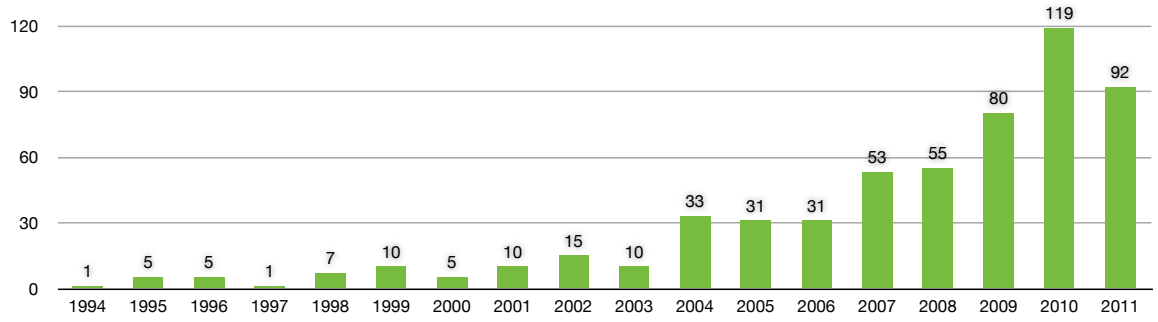

Figure 1: Yearly number of distinct authors contributing to clone research

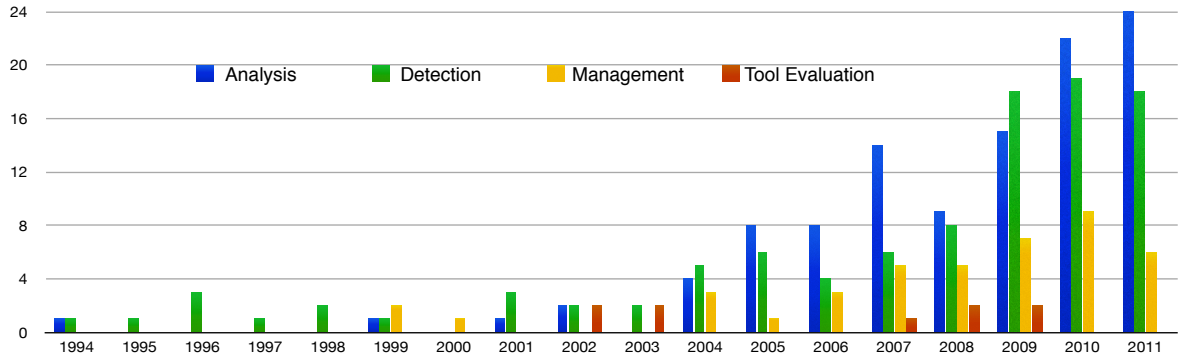

Figure 2: Categories of publications on software clone research in different years

Figure 1 plots the number of distinct authors contributing to clone research in the years from 1994 through 2011. As the figure indicates, the clone research community has experienced a significant growth over the recent years. In the Figure 2, we present the number of publications appeared every year making contributions to each of the four sub-areas of clone research. As seen in the figure, early work on software clone research were dominated by the research on clone detection with some work on analysis. In the recent years, the work on clone analysis and detection has grown significantly while clone management has emerged and growing as a significant research topic. Despite the fast growth of the clone research community, the work on clone management remained much less compared to analysis and detection, which can be more clearly perceived from the Figure 3. This, in combination with the realized importance of research in clone management, point to the further need and potential for research in this sub-area.

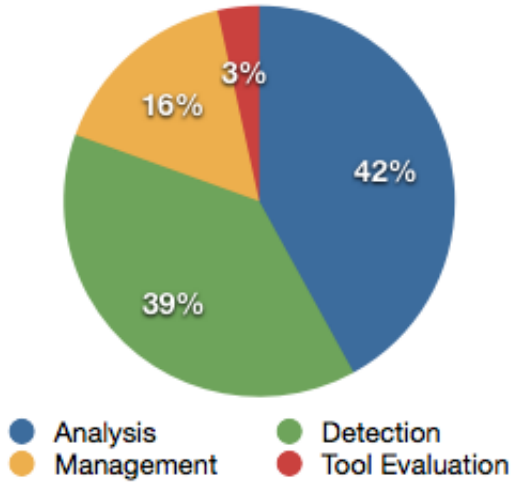

Figure 3: Proportion of publications in each category over the period 1994–2011

It can also be noticed from both the Figure 2 and Figure 3 that over the entire span (1994–2011) of software clone research very few work focused on the evaluation of clone detection techniques or tools, although more than 40 different clone detection tools have been produced realizing a wide variety of techniques [168]. Indeed, the detection of clones is a fundamental topic for software clone research, and the effectiveness of clone management largely depends of clone detection. Therefore, more work on the evaluation of clone detection tools is necessary to inform scopes for further improvements. However, the task can be challenging due to a number of reasons, which we discuss in Section 3.5.2.

2.1 Threat to Validity

The corpus of publications used in our systematic review contains a significant number of scholarly articles published over 19 years. Still, a few publications might have escaped the repository. For example, two M.Sc. theses [5, 171] that were completed at the end of 2011, are yet to appear in the corpus. However, to preserve the reproducibility our systematic review, we did not include such few missing publications. In fact, the corpus is a result of consistent effort for exhaustive preservation of publications, which significantly contributed to the software clones research. Hence, we believe that those few missing articles, if included, would not cause confounding variation in the findings of our review. Indeed, the remaining of this survey is based on relevant literature *beyond* the corpus used in the systematic review.

3 Code Clone

Though, duplicate or similar code fragments are roughly known to be code clones, the definition of clone has remained more or less vague over the last decade. The vagueness is reflected in the definition given by Ira Baxter, "Clones are segments of code that are similar according to some definition of similarity (Ira Baxter, 2002) [125]." Such a definition is dependent on how similarity is defined, and also raises question on how much of code can be regarded as a code segment. Clone research over the past decade somewhat addressed these issues, and the following categorizing definitions of code clone currently have been widely acceptable [125, 163, 161].

Type-1 Clone: Identical code fragments except for variations in white-spaces and comments are *Type-1* clones.

Type-2 Clone: : Structurally/syntactically identical fragments except for variations in the names of identifiers, literals, types, layout and comments are called *Type-2* clones.

Type-3 Clone: Code fragments that exhibit similarity as of *Type-2* clones and also allow further differences such as additions, deletions or modifications statements are known as *Type-3* clones.

Type-4 Clone: Code fragments that exhibit identical functional behaviour but implemented through very different syntactic structure are known as *Type-4* clones.

Indeed, a number of similarity based other taxonomies were also proposed by Mayrand et al. [148], Balazinska et al. [16], Bellon et al. [28, 127], and Davey et al. [42], and Kontogiannis [122].

As described, the first three types of clones are defined based on the similarity in the program text, while *Type-4* clone is defined based on semantic similarity. Thus, *Type-4* clones are also called *semantic clones*. One the other hand, *Type-1* clones are also known as *exact clones*, *Type-2* clones are also sometimes called *renamed clones*, whereas, the *Type-2* and *Type-3* clones jointly are called *near-miss* clones [163]. The terms *parameterized clone* or *p-match clone* are often used in the community to refer to a subset of *Type-2* clones, where there must be a bijective mapping between the identifiers of the two *Type-2* clones [125]. Thus, renaming (arbitrary or systematic) of identifiers are allowed in *Type-2* clones, whereas for parameterized clones, those renaming must be consistent/systemic [163].

For *Type-3* clones, the deletion of statement from one code segment can be considered as an addition of the statement in the other. The difference in the additional or changed statements in the *Type-3* clones are often called the *gaps* [193], and thus *Type-3* clones are sometimes also referred to as *gapped clones* [125, 163, 193]. While the definition of *Type-1* and *Type-2* clones are quite precise, the definition of *Type-3* clones still remains vague [125]. The definition does not precisely indicate how much differences in terms of addition, modification, or deletion of statements are allowed in code segments to be regarded as *Type-3* clones. The practitioners commonly consider code segments as *Type-3* clones when the difference in the statements remain below a (dis)similarity threshold [11, 140, 164, 205]. However, a consensus on an appropriate value for such a threshold is yet to be established [125].

3.1 Clones beyond Source Code

Ongoing research also attempts to deal with clones in software artifacts other than the source code [99], such as clones in higher level code structure [21, 22] or high level concepts [146], clones in the models of formal model based development [46, 47], in UML domain models [180], UML sequence diagrams [142], in the graph based Matlab/Simulink models [160], and duplication in requirement specification documents [99, 100]. However, this paper focuses on the management of clones in the source code only.

3.2 Clone Relationship

A clone relationship exists between two code segments, which are clones to each other according to specification of similarity as prescribed by the definitions stated above. Such a clone relationship is *reflexive* (i.e., if code segment A is a clone of B, then B is also a clone of A). Moreover, for *Type-1* and *Type-2* clones, the *transitive* relationship also exists (i.e., if code segment A is a clone of B, and B is a clone of C, then A is also a clone of C). However, such a transitive property may not hold for *Type-3* clones [27]. The aforementioned definitions also imply that a subset relationship exists among the

Type-1, *Type-2*, and *Type-3* clones. Mathematically, Type-$i \subseteq$ Type-j, for $i \in \{1, 2\}$ and $j = i + 1$ [119].

Two code segments that are clones to each other (i.e., having clone relationship between them) are called a *clone pair*. A *clone-class* or *clone-group* is a set of code segments such that any two of them are clone pairs. The term 'clone' is used in the software community in the following two ways.

'clone' as a noun refers to a code fragment that is, according to the aforementioned definitions, similar enough to one of more other code segments. For instance, our objective is to find all *clones* in the code-base.

'clone' as a verb indicates the act of producing a code segment (e.g., by copy-pasting) that is, according to the aforementioned definitions, similar enough to one of more other code segments. For example, I *cloned* the function to reuse it in my context.

3.3 Clone Granularity

The definitions of the all four types of clones are based on the notion of code segment, but how much of contiguous code can be considered as a code fragment is not made specific. Thus, contiguous portion of code at different levels of granularity have been used in the literature. As concerned with source code, the most commonly used granularities are at the level of the entire source file, class definition, method body, code block, and statements, which yield the the notion of code clones of the following five types:

File clone: When two files are found to have contained similar enough source code, they are called file clones.

Class clone: Two classes of an object-oriented code can be considered as class clones if they have identical or near identical code.

Function clone: Two functions are considered as clones when the bodies of the functions consists of code that are similar enough.

Block clone: When two blocks of code (marked with opening and closing braces or indentation, or the like) are similar enough, they are called block clones.

Arbitrary statements clone: When two groups of statements at arbitrary regions of the source file are found to be similar enough, they are also regarded as clones (`CCFinder` detects such clones).

3.3.1 Which Level of Granularity Is Appropriate?

According to the definition of clone, a pair of very small portion portion of code such as, two identical identifiers, two similar statements, functions or blocks each having only one statement can also be valid candidates for clones. However, those tiny code segments cannot be real clones of pragmatic significance. Thus, the practitioners typically disregard those code segments that are smaller than a given threshold. Again, due to the differences in the varying contexts and techniques for clone detection, there has been no consensus in the community on such a threshold, though minimum 20 to 30 tokens [118, 172] or three to five lines of code [167, 206, 203, 205] is a common threshold used in practice.

We believe that the chosen granularity of the code segments should exhibit some characteristics so that the detected clones at that granularity actually becomes useful from the maintenance perspective. In this regard, we propose the following desired characteristics as inspired by our experience and the criteria proposed by Giesecke [62].

Coverage: The set of all code segments should cover maximal behavioural aspects of the software.

Significance: Each code segment should possess implementation of a significant functionality.

Intelligibility: Each code segment should constitute sufficient amount of code such that a developer can understand its purpose with little effort.

Reusability: The code segments should feature a high probability for informal reuse.

A source file typically contains a large bulk of code (a Java source file may contain multiple classes, interfaces, and so on). Though, the set of all source files cover all behavioural aspects of the software, informal reuse at file level is unlikely in a software system. Classes in object-oriented systems also satisfy the *coverage* criterion, but they are also unlikely to be informally reused, as more elegant concepts such as inheritance and delegation are there for their formal reuse.

Methods or blocks (a method body itself forms a block) cover almost all behavioural aspects, and those that are missed (e.g., library/package inclusion, declaration and initializations) can be neglected [62]. Methods and blocks contain implementation of significant functionality that can be understood with less effort than to do for an entire source file or class. Methods isolate a functionality and so often do the blocks, and thus they are likely to be informally reused, as there is no easy rigorous way to reuse methods from another context [62].

Code segments at the arbitrary statement level satisfy the *coverage* criterion, but not the other three criteria. Single statement or short sequence of statements typically

don't cover a unit of significant functionality. The meaning of a statement in general is likely to be incomprehensible without considering its context, and the context of the host block or function. Moreover, sequences of import/include statements, declaration/initialization statements, or the sequences of statements spanning code boundaries such as boundaries of two functions, classes, or blocks do not exhibit significant potential for reuse, rather those should be ignored in many cases.

Thus, we suggest that code segments at the level of functions or blocks can be the most suitable granularity for dealing with code clones, specially for maintenance. Giesecke [62] also proposed in favour of function as the most adequate level of clone granularity. Indeed, block level granularity also covers functions, though it does not distinguish a block that constitutes a function body from that which does not. Driven by the same understanding, Higo et al. developed `CCShaper` [79, 76] to post-process the clone detection result from `CCFinder` to extract block level clones as the potential candidates for refactoring.

3.4 Intentional and Accidental Clones

Code clones in a software system can appear in two ways. First, the programmer copies an existing code, pastes it in another place, and thus reuses the implementation with or without further modifications. The resulting code may still remain similar to the original and form a clone-pair. Such clones created by the programmer's deliberate copy-paste-modification activities are known as *intentional clones* or *copy-pasted* clones. There are a number of reasons why programmers create such intentional clones. An obvious reason is to reuse existing implementation without "re-inventing the wheel".

Indeed, similar code segments may also appear in the system without the programmer's intention, and often the developer can even be unaware of the creation of such clones. For example, due to the developer's mental model, frequently used idioms are reproduced from memory instead of deliberate copy-paste [161]. Even different developers may also produce very similar code while solving a similar problem, or due to the dictation of certain APIs they use, or using the same design pattern [203]. Such similar code snippets that are not produced by deliberate copy-paste operations are known as *unintentional clones* or *accidental clones*.

Practically, there are many reasons behind the creation of intentional and accidental code clones in a software system. Further details on the root causes of cloning can be found elsewhere [109, 110, 125, 163].

3.5 Clone Detection Techniques

Over more than a decade of code clone research a number of techniques have been devised for the detection of code clones and many clone detection tools have been

developed. Prominent techniques can be categorized as shown in the Table 2. The categorization is based on the type of information used in the analysis/comparison and the type of approaches used for the analysis. The first technique (Tracking Clipboard Operations) in the table is an action-trace based technique and the rest others are code similarity based techniques. In this section, we provide a brief summary of different clone detection techniques, and point out the strengths and weakness of those techniques in general. More detailed descriptions of those techniques can be found in the corresponding papers and elsewhere [163].

Tracking Clipboard Operations: This technique of clone detection is based on the assumption that programmers' copy-paste activities are the primary reason for the creation of code clones. So, the technique simply tracks clipboard activities in the editor (inside IDEs such as Eclipse) when a programmer copies a code segment and reuses by pasting it. The copied and the pasted code segments are recorded as clone-pairs.

Metrics Comparison: Metrics based techniques are usually used to detect function clones. The techniques are based on the assumption that similar code fragments should yield very similar values for different software metrics (e.g., cyclomatic complexity, fan-in, fan-out). Typically, for the code segments a set of metrics are gathered into vectors. The differences in the vectors are calculated, where the close vectors (e.g., measured by Euclidean distance) indicate that their corresponding code fragments are clones.

Texual Comparison: Text based techniques compare program text, typically line by line, with or without normalizing the text by renaming the identifiers, filtering out the comments and differences in the layout.

Token Based Comparison: The entire program is transformed into a stream of tokens (i.e., individual units/words of meaning) through lexical analysis. Then the token stream is scanned to find similar token subsequences, and the original code portions corresponding to those subsequences are reported as clones.

Syntax Comparison: Syntax comparison based techniques are developed on the fact that similar code segments should also have similar syntactic structure. Thus, the program is parsed to produce syntax tree, where the similar subtrees indicate that their corresponding code segments are clones.

PDG Based Comparison: For a given program, a set of PDGs (Program Dependency Graphs) [57] are produced based on the data and control dependencies among the statements of the program. The code segments corresponding to the isomorphic subgraphs are identified and reported as clones.

Comparison of Low Level Form of Code: Instead of analyzing and comparing textual source code, the techniques analyze the lower level code (e.g., assembly code, Java bytecode) as obtained from the transformation done by the compiler.

Other Techniques: Beside the aforementioned prominent techniques for clone

Table 2: Code Clone Detection Techniques (extended from [126])

Tracking Clipboard Operations	
copy-pasted code segments are recorded as clones	[37, 45, 83, 197]
Metrics Comparison	
comparing metrics for functions	[123, 124, 132, 148]
comparing metrics for web sites	[48, 133]
Textual Comparison	
hashing of strings per line, then textual comparison	[95, 96]
hashing of strings per line, then visual comparison using dot-plots	[52]
latent semantic indexing for identifiers and comments	[146]
syntactic pretty-printing and normalization, then textual comparison between lines	[164]
syntactic pretty-printing and normalization, then comparison of fingerprinted lines using suffix tree	[205]
syntactic pretty-printing and normalization, then hash based comparison of functions/blocks	[190]
Token-based (Lexical) Comparison	
suffix trees for tokens per line	[10, 11, 12]
token normalizations, then suffix tree/array based search	[23, 105, 112]
data mining for frequent token sequences	[140]
Syntax Comparison	
hashing of syntax trees and tree comparison	[25, 198, 8]
data mining for frequent syntax subtrees	[194]
serialization of syntax trees and suffix-tree detection	[55, 127, 138]
derivation of syntax patterns and pattern matching	[54]
metrics for syntax trees and metric vector comparison	[91, 154]
PDG Based Comparison	
approximate search for similar subgraphs in PDGs	[61, 78, 81, 120, 128]
Comparison of Low Level Form of Code	
comparing Java bytecode	[13, 175]
comparison of compiled (assembler) code	[43, 44, 170]

detection, other techniques, such as anti-unification [35, 138], formal methods [175], and combination of distinct techniques [60, 137] were also approached. Tracing of abstract memory states during the execution of the program was also attempted to detect semantic clones [116].

3.5.1 Strengths and Weaknesses of Clone Detection Techniques

Due to the orthogonal nature of the different approaches for clone detection, it is inappropriate rate one technique over another. Each of the techniques have their strengths and weaknesses. In this section, we briefly discuss the strengths and weaknesses of different clone detection techniques. More comprehensive qualitative evaluation of the existing clone detection techniques and tools can be found elsewhere [165, 168].

Clone detection using *clipboard operations (copy-paste)* has the benefit that clones are captured at the time of their creation. However, the technique poses a number of questions and limitations towards pragmatic design decisions to be made for realizing the technique in a tool:

- How much (granularity) of copy-pasted code should be recorded as clones?
- Should the copy-pasted code that spans beyond a syntactic block be considered as clone?
- Copy-pasted code may later be modified and become very different from original. Should the very dissimilar code segments still be treated as clones (as done in `CloneScape` [37]), or some similarity based decision has to be made?
- Since the tracking of copy-paste activity is coupled with a certain editor, the technique is vulnerable to changes in the source code outside the editor.
- The technique cannot deal with unintentional/accidental clones, or clones in legacy systems.

The effectiveness of *metrics based* techniques depends on the selection of a broad set of orthogonal significant metrics. Typically, metrics based techniques may be computationally expensive for the overhead of calculating the chosen metrics. Moreover, due to the fact that different code segments may, at times, have the same value for certain metrics, the metric based techniques may report many false positives.

The advantage of simple *textual comparison* is that the technique, in general, is independent of programming languages and the program does not need to be complete of syntactically correct. However, the technique (without advanced normalization and filtering of differences in layouts) may susceptible to even minor changes in the source code [126]. Due to the use of lexical analysis only, the *token based* techniques also can operate on incomplete or syntactically incorrect program. But, the token based

techniques may detect code clones that span beyond the boundaries of syntactic blocks.

The techniques based on *syntax comparison* suffer from the overhead of invoking a parser to generate parse tree. Due to the use of a parser, the technique is coupled with programming language syntax. The parser based syntax comparison techniques are very accurate in detecting clones that have similar syntactic structure. Still, such techniques are vulnerable to even subtle differences in the nesting of code, so may fail to detect many potential clones. Earlier empirical evaluations [28, 27, 9] also suggest low recall of syntax comparison based clone detection techniques.

PDG based techniques analyze the source code at a level higher than that of syntax comparison based techniques, and thus can capture the program semantics to some extent. PDG based techniques can detect clones consisting of non-contiguous code. However, producing PDGs is computationally expensive. In addition, finding isomorphic subgraphs from PDGs is an NP-hard problem [126]. Therefore, approximation algorithms are used, which do not aim for the optimum. The PDG based approach is tolerant to reordering of certain program elements, which is both a strength and weakness of the technique. The ordering of statements specify the execution path of a program written in traditional imperative language, but the PDG based techniques often disregard such order and reports clones composed of *non-contiguous* code segments, there remains quite good chance for false positives in detecting clones of practical significance.

Clone detection in the compiled code (e.g., assembly code, binary executable, Java bytecode) requires the source code to be complete and syntactically correct. Moreover, compilation generates the lower level code that often normalizes syntactic variants of source code into a compact canonical representation free from comments and differences in the layouts as in the source code. This, in principle, should make it easier to capture semantically equivalent constructs that might "look different" in the source code written in a high level language [43]. On the contrary, small difference in the source code may produce very different sequences of bytes in the compiled code, and thus finding code clones based on byte code similarities may be even more difficult than finding clones in source code [13]. Moreover, clone detection based on the analysis of compiled code is typically performed at class level, because distinguishing the portion corresponding the functions, blocks or other level of granularity and mapping those back to locations in the higher level source code can be very challenging.

The purpose of clone detection techniques is not limited to only clone management for improved software maintenance. The variable strengths of different clone detection techniques lead to their applications in solving diverse research problems such as, plagiarism detection [31, 74], program fault localization [26, 36, 89, 92, 103], malware detection [34, 196], finding crosscutting concern code [33], aspect mining [113, 32],

quality assessment of requirements specification [101], and more. Indeed, the choice of an appropriate clone detection technique largely depends on the context and the purpose [126].

3.5.2 Challenges in the Empirical Evaluation of Clone Detection Tools

Typically, the effectiveness of clone detection is measured in terms of precision (p) and recall (r),

$$p = \frac{|C_s \cap C_d|}{|C_d|}, \qquad r = \frac{|C_s \cap C_d|}{|C_s|}$$

where C_d denotes the set of clones that a particular clone detection tool identifies, and C_s refers to the set of clones actually existing in the code base. But, how can one accurately find all the clones in a system? A reference corpus is necessary to serve as the set of all clones in a system. To obtain this, manual investigation can be a solution for small systems, but the time and effort needed for such a subjective analysis is likely to be impractical for a fairly large system. However, the reference corpus needs to be large to be able to evaluate the scalability of the clone detector under consideration. Therefore, in practice [190, 205], one or more clone detectors that are already known to be effective, are applied to detect clones from a system, and the union of all the sets of clones reported by those tools, is used as a reference corpus to examine the precision and recall of the clone detection tool to be evaluated.

Using a similar idea, Bellon et al. [28] carried out a case study to compare the effectiveness of six clone detectors (`CCFinder, CloneDr, Dup, Duploc, Duplix` and `CLAN`) representing different clone detection techniques. Bellon's study [28], which was performed about six years ago, is still the most comprehensive quantitative study on the comparative evaluation of clone detectors, and the reference benchmark produced from the study is the only notable benchmark data available to date [126]. A similar but smaller scale study was also conducted earlier by Bailey and Burd [9].

Bellon's benchmark data was constructed as the union of the sets of clones reported by the clone detectors subject to the study. Then the precision and recall for each tool was computed based on how many of the benchmark clones a particular tool detected. Thus, the result was skewed in favour of those tools that contributed more in the construction of the benchmark data. Moreover, only 2% clones of the reference set were manually verified. Hence, there is a possibility that the benchmark data might have included significant amount of false positives that could have been reported by those participating tools.

The Need for Reference Benchmark: The above discussion suggests that, a reliable large set of benchmark data[1] is still necessary, and new tools should be evalu-

[1]Similar to what available at `http://math.nist.gov/MatrixMarket/` used in comparative studies of algorithms for numerical linear algebra.

ated with the standard benchmark data before publication [125]. Both precision and recall should be reported, as precision or recall alone draws a partial picture only. Moreover, both the runtime complexity and memory efficiency should be reported, since these two criteria affect the scalability of a certain technique. Theoretical runtime complexity only can be misleading, because high memory requirement may cause time-consuming page swapping at the operating system level, and increase the execution time in practice [169].

The creation of a reliable large benchmark data set with manually verified clones can be a mammoth task. The tedious work of manual investigation and verification can be accomplished by collaborative efforts from the community. The *mutation framework* [166] proposed by Roy and Cordy can also be used for the purpose by injecting a set of artificially created (and manually verified) clone fragments in various locations of a certain code base, and the set of synthetic clones can serve the purpose of a reference benchmark. Indeed, the vagueness in the definitions of clones may lead to disagreement in different human evaluators' perception of clones, as found in the study of Walenstein et al. [195]. Therefore, the purpose of the benchmark data should be defined first, and the definition of similarity should be formalized accordingly, as well as the level of clone granularity. A purpose could be, for example, the evaluation of clone detection techniques for clone maintenance and removal.

3.6 Clone Evolution

Software development and maintenance in practice follow a dynamic process. With the growth of the program source, code clones also experience evolution from version to version. Many studies have been conducted to date for understanding the overall evolution [4, 3, 143, 65, 206], stability of cloned code [14, 66, 82, 130, 131, 151], the relation of clone evolution with software faults [19, 69, 179], and other characteristics of clone evolution. While the high studies inform the characteristic and impact of code cloning, more lower level analyses that investigate the change patterns in the evolution of individual clone fragments can suggest techniques for optimizing clone management including refactoring and removal. However, there is a recent survey on clone evolution [159]. Hence, we keep this section brief with specific focus on the evolution of individual clone fragments and their change patterns.

3.6.1 Clone Genealogy

Kim et al. [117, 118] first coined the term *clone genealogy*, which refers to a set of one of more lineage(s) originating from the same clone-group. A *clone lineage* is a sequence of clone-groups evolving over a series of versions (revision or release) of the software system. Thus, a clone genealogy (Figure 4) describes the evolution history of a particular clone-group over subsequent versions of the system. The extraction

of clone genealogies from a series of versions of a program has been identified as the fundamental task to study the evolution of individual clone-groups.

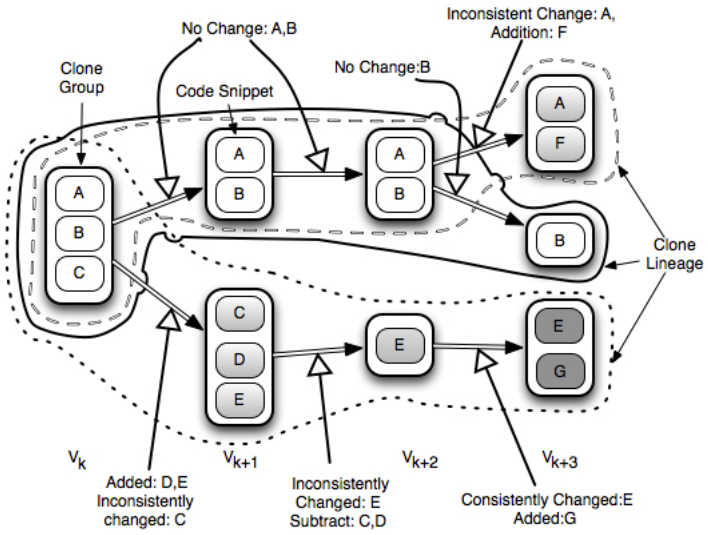

Figure 4: A clone genealogy with two lineages over versions v_k through v_{k+3} [201]

Genealogy Extraction: To map clones across subsequent versions of a program (i.e. extraction of clone genealogies) mainly four different approaches have been found in the literature, as summarized in Table 3. While most of these approaches [19, 14, 118, 172, 7] focused on genealogies of *Type-1* and *Type-2* clones, `gCad` [171, 173] is the only *Type-3* clone genealogy extractor to date released as a separate tool.

3.6.2 Clone Change Patterns

Recent genealogy based studies [118, 172, 173] on clone change patterns characterized the evolution of a particular clone-group based on the following four categories of transitions between subsequent versions of the software system.

Addition/Grow: One or more clone fragments is added to the clone-group in the next version.

Deletion/Shrink: One or more clones disappeared from the clone-group in the next version.

Consistent Change: All clone in the clone-group experience the same set of changes during transition to the next version.

Table 3: Summary of techniques for clone genealogy extraction

Approach	Strength	Weakness	Citation
Separate clone detection from each version, and then similarity based heuristic mapping of clones in pairs of subsequent versions	flexible	quadratic runtime [70], susceptible to large change in clone	[14, 118, 172]
Clones detected from the first version are mapped to consecutive versions based on change logs obtained from source code repositories	faster than the above technique	can miss the clones introduced after the first version [171]	[7, 19, 129, 187]
Clones are mapped during the incremental clone detection that used source code changes between revisions	faster than the above two techniques	cannot operate on the clone detection results obtained from traditional non-incremental tools [171]	[68, 155]
Separate clone detection from each version, functions are mapped across subsequent versions, then clones are mapped based on the mapped functions	improved runtime	susceptible to similar overloaded/overriden functions	[145, 171, 173]

*A combination of the first and second approaches was also used in some studies [29]

Inconsistent Change: During transition to the next version, at least one clone in the clone-group experiences changes that are inconsistent with changes in other member(s) of the clone-group.

Same/No Change The clone-group moves to the next version without experiencing any change in the members.

On the basis of the aforementioned categorization, clone genealogies are classified as follows:

Static Genealogy: A genealogy where the clones in the clone-group do not experience any change over the series of subsequent version.

Inconsistently Changed Genealogies: A genealogy where one or more clones in the clone-group experience inconsistent changes during transition between any two subsequent versions.

Consistently Changed Genealogies: A genealogy where none of the clones in the clone-group experience inconsistent changes during transition to subsequent versions.

Dead Genealogy: A genealogy that does not survive till the last version (i.e. before the last version the clone-group disappears).

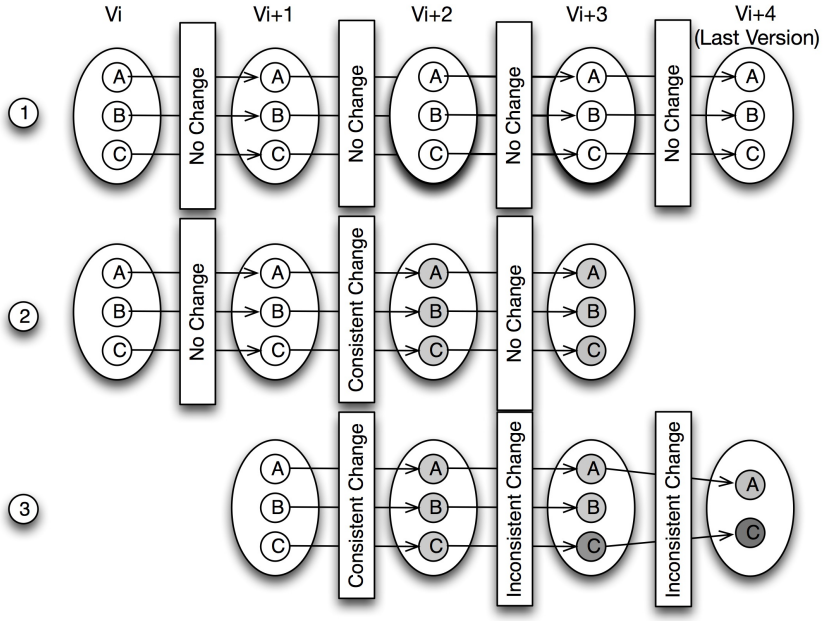

Figure 5: Clone change patterns and types of genealogies [171]

Alive Genealogy: A genealogy that survives at the last version of the software system.

Aversano et al. [7] further caracterized inconsistent change pattern into two categories:

Independent Evolution: In independent evolution, the clones of a clone-group, once changed inconsistently, evolve independently across versions. Thus, independent evolution may cause branching with multiple lineages in a genealogy.

Late Propagation: In late propagation, one or more clones in a clone-group may experience inconsistent changes and may disappear from the clone-group in the next version, but at a later version those clones re-appear in the clone-group.

Studies [14, 29] on clone change patterns revealed that inconsistent changes in clones sometimes caused program faults. Moreover, late propagation is reported to have even more significant correlation with software defects and thus concluded to be "more risky than other clone genealogies" [19].

Using the clone genealogy model, recently Saha [171] studied the change patterns in the evolution of clones in five open-source software systems. Major findings from the study report some common properties of frequently changed clone-groups, such as, cohesive clone-groups having small number of clones experienced more changes than

others. Intuitively, such clone-groups can be easier for modification/refactoring and it is unlikely that the developers of those open-source systems used any tool support dedicated for clone management. Thus the clone changes could have exhibited a very different pattern, had the developers used proper clone management tool. The frequent changes to small cohesive clone-groups rather suggests the necessity of tool support for dealing with specially the dispersed (or diffused [59]) large clone-groups.

3.6.3 Need for Improvements

Studies on clone change patterns using the genealogy based model described above can suffer from a number of issues. First, due to the threshold based similarity measure used in practice, specially for *Type-3* clones, there remains an open question on the appropriate value for the threshold. Moreover, for *Type-3* clones, is it an appropriate practice of grouping *Type-3* clones into non-overlapping sets? If not, the traditional notion of genealogy cannot not apply to *Type-3* clones. Can we devise a more appropriate alternative?

Second, a genealogy can be characterized as inconsistently changed if only a single clone over the entire length of the genealogy experiences even a very minor inconsistent change. To draw a better picture, we may capture formation such as, what portion of clones in clone-groups change in how many versions, how large the changes are.

Third, the definitions of dead/alive genealogies in terms of the *last versions* are subject to the choice of versions one makes to study. Different choice in the number of versions would have different last version. Moreover, consider a genealogy originated at the second last version and extends to the last version, while another genealogy originated from one of the very early versions and extended to the second last version and at the last version it disappears. The current genealogy model will categorize the first genealogy as alive, while the second as a dead genealogy. Thus, we are missing important information about how long the genealogies propagate. Hence, instead of characterizing as alive/dead genealogies, we might characterize genealogies in terms of what portion of the series of versions the genealogies span, similar to the notion of *succession-length* of Zibran et al. [206].

Finally, from the correlation between late propagation and software defects, can we really derive a causal relationship concluding that late propagation is riskier than other clone genealogies? Inconsistence changes are believed to often cause defects, and clones may disappear from a genealogy due to inconsistent changes. Later modifications, which could be even bug fixing activities, may cause changes in the disappeared clone to sync it to its original clone-group. In such a scenario, late propagation actually contributes in repairing the defect introduced from inconsistent changes. Thus, we believe, late propagation can really play a dual role and more studies are necessary to distinguish them.

3.6.4 Visualization of Clone Evolution

Visualization support can aid the analysis of clone evolution, and thus different techniques and tools have been proposed for visualizing properties of clone evolution including the genealogy model.

Adar and Kim [2] developed `SoftGuess`, a system for clone evolution exploration that supports three different views. `SoftGUESS` is developed on top of `GUESS` [1], the graph exploration system which models the evolution of a software system using graphs. The genealogy browser offers a simple visualization of clone evolution where nodes represent clones, arranged from left to right, and the those that belong to the same class are arrangedvertically in the same position. Thus, each column represents a version. Link between a pair of node reflects the predecessor and successor relationship during the evolution of the software. The encapsulation browser shows how clones within a clone-group are distributed in different parts of a system and how they fit in the hierarchical organization of the software system by visualizing the containment relationship through a tree structure. Finally, the dependency graph describes how the nodes (package, class or method) within a version are evolved from other nodes and how they evolve in the next version. In addition, `SoftGUESS` also supports charting and filtering mechanisms based on Gython, a SQL type query language. However, `SoftGUESS` lacks an 'overview' feature and requires users interaction for data reduction through queries. Although a query is a powerful mechanism to identify important patterns of cloning, formulating query could be difficult as this requires more cognitive effort from the developers.

Recently, Harder and Göde [72] developed a multi-perspective tool for clone evolution analysis, called `CYCLONE`. It offers five different views to analyze clone data stored in a RCF file, where RCF is a binary format to encode clone data including the evolutionary characteristics. The evolution view in `CYCLONE` visualizes clone genealogies that uses simple rectangles and circles to denote software entities. Each circle represents a clone fragments arranged in a set of rows where each row represents a particular version of the software. The clones that belong to the same clone class are packed within a rectangle. Finally, lines represent the evolution of a clone fragments. In addition, the view employs colors to distinguish types and the changes of the clones. Although the view highlights many important evolutionary characteristics, the volume of data produced by the genealogy extractor still limits its usefulness, thus call for overview and filtering mechanisms. A similar visualization support is available in `VisCad` [5], with additional flexibility of metric based filtering of genealogies.

Saha et al. [174] presented an idea for clone evolution visualization using the popular scatter plot. In their proposed approach, scatter plots show the clone pairs associated within a pair of software unit (file, directory or package). Based on the type of clone genealogies they are associated with, clone pairs are rendered with

different colors. Selecting a clone pair through user interactions (double clicking on a clone pair in the scatter plot) shows the associated genealogy in a genealogy browser. The proposal facilitates developers or maintenance engineers to identify evolutionary change patterns of the clone classes in a particular version and then provide a way to call for genealogy browser to dig deeper. However, it does not provide overall characteristics of the genealogies, and neither an implementation of the proposal nor an empirical evaluation of the technique is available yet. Moreover, due to the large number of clone pairs, selection and useful pattern identification in such a scatter plot can be difficult, which is why different variants of the traditional scatter plots appeared in the literature [40].

4 Clone Management

"Clone management summarizes all process activities which are targeted at de- tecting, avoiding or removing clones" [62]. Thus, clone management encompasses a broad categories of activities including clone detection, tracking of clone evolution, and refactoring of code clones. As support for these operations, the documentation and analysis of code clones can be regarded as parts of clone management. Moreover, clone visualization may also be an effective aid to clone analysis, per se, clone management.

4.1 Clone Management Strategies

For dealing with code clones, Mayrand et al. [147] proposed two concrete activities namely "problem mining" and "preventive control", which were further supported by a later study of Lague et al. [132]. Giesecke [62] categorized them into compensatory and preventive clone management, respectively. Giesecke [62] suggested that all clone management activities can be associated with on or more of the three categories: corrective, preventive, and compensatory management.

Corrective clone management aims for *removal* of existing clones from the system. The objective of *Preventive clone management* is to prevent creation of new clones in the system. *Compensatory clone management* deals with applying techniques (such as annotation, documentation) for avoiding the negative impacts of clones that are not removed from the system for some valid reasons. In practical settings, avoiding clones may be impossible at times, and the expectation of a clone-free system can be unrealistic. Thus, *preventive* clone management actually refers to *proactive* management [83, 84] that aims to deal with the clones during their creation or soon after they are introduced. An opposite strategy, *retroactive clone management* [37] adopts the *post-portem approach* [205], where clone management activities initiate after the development process is complete up to a milestone.

Clone management in legacy systems can be the most appropriate for the *post-mortem* strategy. Indeed, prevention is easier to cure. Therefore, *proactive* clone management is preferable to *post-mortem* approach. While, ideally, all clones should be managed proactively, in practical settings, proactive treatment to all clones may not be feasible or possible. Therefore, a versatile clone management system should focus on the support for proactive management, while at the same time, should also facilitate retroactive clone management [37].

4.2 Design Space for a Clone Management System

Most of the clone detectors [105, 164, 91, 78] out there are implemented as stand-alone tools separate from IDEs (Integrated Development Environments) and typically searches for all clone in a given code base. While clone detection from such tools can help clone management in post-mortem approach, researchers and practitioners [62, 71, 83, 84, 132, 154, 204, 205] believe that clone management activities should be integrated with the development process to enable proactive management.

Hou et al. [84], during the on-going development of their clone management tool `CnP` [83], explored the design space towards tool support for clone management. However, their work was tightly coupled with the clone detection technique based on the programmers' copy-paste operations. Thus, their findings are limited in scope to the management of copy-pasted code, and most of the findings are not applicable to clone management based on similarity based clone detection.

We identify three major dimensions and some sub-dimensions in the design space for a versatile clone management tool. These dimensions are inspired by our experience and the different clone management scenarios reported by Giesecke [62].

4.2.1 Architectural Centrality

The need for the integration of clone management activities with the development process suggests that the IDEs should include features to support for clone management activities during their actual development phase. While a programmer typically works inside an IDE running on her individual workstation, for fairly large projects, specially in industrial settings, a team of developers collaboratively work on a shared code base kept in a version control system (e.g., SVN, CVS) set up on a central server. Hence, the supports for clone management activities can be implemented as features augmenting the local IDEs, or the functionalities can be implemented at the central repository.

Decentralized Architecture: The clone management functionalities, when augment the features in local IDEs, can enable the individual programmers to exploit the benefit of clone management. In the decentralized scenario different developers can use different tools, and some programmers can get the flexibility to completely or

partially disregard clone management at their respective situations. Apparently, such a decentralized implementation may completely disregard the existence of a central server, and enforces proactive clone management before check-in to the shared repository. However, this necessitates additional requirement for establishing means for communication between distributed developers, as well as combining and synchronizing clone information across all the developers.

Centralized Architecture The centralized architecture inherently aims to support clone management in distributed development process. The functionalities can be implemented as a client-server application on top of the central version control systems. Such a centralized clone management system may require greater effort and offer less flexibility than a decentralized implementation [62]. Indeed, a client-server implementation cannot support those individual programmers who work alone on their stand-alone local machines [205].

4.2.2 Triggering of Clone Management Activity

A clone management activity may be initiated by the practitioner, or such an activity may be triggered in response to certain actions in the system. Thus, an activity can be human triggered or system triggered.

Human Triggered Initiative: A developer, after writing or modifying a piece of code, may invoke search for its clones in the system, and upon finding the clones, she may analyze and decide how to deal with them. In such an ad-hoc triggering scenario, the developer, at times, may forget to perform the necessary clone management. An instance of clone management activity may also be periodically scheduled in advance as part of a larger plan of process activities, and clone management activities can be carried out following the post-mortem approach on the current status of the code base.

System Triggered Initiative: The development environment can trigger clone management activities in response to certain events, such as saving changes in the code, or check-in of modified code to the central repository. Such events may notify and suggest the developer to perform the required clone management operations. However, care must be taken so that those auto-generated notifications and suggestions do not irritate the developer or hinder her normal flow of work.

4.2.3 Scope of Clone Management Activity

An instance of clone management activity may be *clone-focused* or *system-focused*. A clone-focused activity deals with a narrow set of clones of a particular code segment of interest. On the contrary, a system-focused clone management activity aims to deal with broad collection of clones in the entire code base, or particular portions of the system.

4.3 Clone Management Activities

To manage clones, first they have to be identified. The result of clone detection forms clone documentation that records the location of code segments and their clone relationship. If the code base changes due to ongoing development, the changes and locations of the clones need to be tracked, and the documentation needs to be updated accordingly. The clone documentation may be analyzed to determine justification of clones or to find potential clones for removal. Visualization techniques can aid such analysis. Clones that are found to have justified reason to exist, may be further documented and/or annotated. The candidates for refactoring can be scheduled for modification and/or removal. Upon the application of refactoring operations, a follow up verification may examine if the refactoring caused any change in program behaviour, and in accordance, may initiate roll back and re-refactoring. Upon completion of refactoring the clone documentation needs to be updated for consistency.

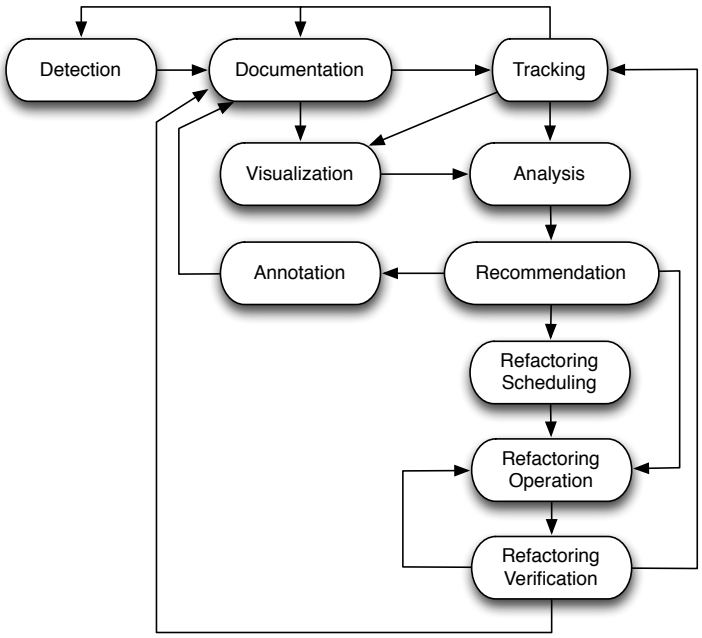

Figure 6: Clone management workflow

The workflow for a typical clone management system may compose all these activities according to as summarized in the Figure 6. In the following sections, we describe the state of the art in support for each of the clone management activities.

4.4 Integrated Clone Detection

There are many clone detection tools our there, each has its own strengths and weaknesses. However, for proactive clone management, the support for clone detection should be integrated with the development process. Therefore, we focus on those tools that realized the clone detection feature integrated with an IDE or a version control system.

Juergens et al. developed `CloneDetective` [102], an open source framework to facilitate implementation of customized clone detectors. `CloneDetective` separates distinct phases of typical clone detection process and offers skeletal implementations of those phases accompanied by provision for declarative configuration facility for customizing the clone detection approach. The framework also offers a separate stand-alone viewer for visually present the result of clone detection. The viewer can be integrated with the Microsoft Visual Studio.NET or Eclipse IDE. The framework itself is built on the infrastructure of `ConQAT`[2], an integrated toolkit for software quality assessment. However, beyond the detection of clones and visualization of the clone detection result, they offer no further support for clone management. Moreover,the suffix-tree-based implementation of clone detection approach can detect Type-1 and Type-2 clones but "probably not" Type-3 [168].

`SimScan`[3], which is a parser-based tool available as plugin to Eclipse, IDEA, or JBuilder, can detect *Type-1*, *Type-2*, and possibly a subset of *Type-3* clones. The potential of `SimScan` is also limited up to the detection code clones, not beyond that.

Giesecke [62] proposed a generic model for describing clones. The model allowed separation of concerns among the detection, description, and management of code clones. The objective was to ease the implementation of tools to support such activities. Based on the proposed model, they implemented `DupMan`, a framework [63] integrated with the Eclipse platform, and developed a prototype tool having `SimScan` as the back-end clone detector. The model and implementation also limited in the detection of clones and the representation of the clone information for persistence.

Chiu and Hirtle developed `CloneScape` [37], a plugin to the Eclipse IDE, that can detect and track code clones based on the copy-paste clip-board activities in the editor. They proposed a number of views based on heated scatter plot and graph views to help clone visualization and analysis. However, other than the detection and visualization of clones, `CloneScape` offers no further support for clone management. Moreover, the implementation of the tool (specially the visualization part) is incomplete and there is no evaluation of the effectiveness and usability of their approach.

`CloneBoard` [45] and `CPC` [197] are Eclipse plugins similar to `CloneScape` that can detect and track clones based on clip-board (copy-paste) activities of program-

[2] `http://www.conqat.org/`

[3] `http://blue-edge.bg/download.html`

mers. Both CPC and CloneBoard support linked editing of clone pairs as described by Toomim et al. [188]. However, CPC was implemented as a framework to serve as a platform for future clone management technology, whereas, the focus of CloneScape was more on clone visualization and navigation, though their implementation remained incomplete. Hou et al. are developing a toolkit named CnP [83] for clone management, which also detects clones based on programmers' copy-paste activities. Indeed, the current implementation of CnP offers very limited support for clone management, which we address in Section 4.8.4.

SHINOBI [112] is an add-on to the Microsoft Visual Studio 2005. For clone detection, it parses the source code, extracts sequence of pre-processed tokens and creates an index using suffix array technique. SHINOBI internally uses CCFinderX's preprocessor, and thus it can detect *Type-1* and *Type-2* clones only, but not *Type-3* [205]. It was developed as a client(IDE)-server(CVS) application to mainly relocate the the clone detection overhead from the client to a central server. It simply displays clones of a code fragment underneath the mouse-cursor, no further support for clone management is offered. CodeRush[4] is a commercial add-in to the Microsoft Visual Studio for providing assistance in coding and refactoring. CodeRush version 11.2 recently introduced a new module DDC for the detection and consolidation of duplicated code.

Li and Thompson developed Wrangler [138], a tool for supporting refactoring of functional programs written in Erlang. Wrangler can be integrated with Emacs or Eclipse IDE through the ErlIDE plugin. Wrangler can detect clones from Erlang programs in several phases. First, the program is parsed to generate AST. Then the AST is normalized (generalized) by replacing certain expression by placeholders. The source code corresponding to the generalized AST is then pretty-printed and serialized into a single sequence of delimited expressions. Each expression statement is then hashed to generate a sequence of hash values, which are then used to build a suffix tree. The clones detected from the suffix tree are then examined through anti-unification for filtering out the false positives. Thus Wrangler can detect *Type-1* and *Type-2* clones only, but probbaly not *Type3*. In addition, the clone detection approach of Wrangler is computationally expensive. The anti-unification technique for the examination of a single clone-group having n members itself has the $O(n^2)$ worst case runtime complexity [138]. Wrangler also offers limited support for a few clone management activities, which are addressed in Section 4.8.5 .

Bahtiyar developed JClone [8] as a plugin to the Eclipse IDE for detecting code clones from Java projects. JClone applies an AST based technique to detect *Type-1* and *Type-2* clones only. It enables the user to trigger the detection of clones from one or more selected files or directories. It also offers a few visualization (i.e., TreeMap and CloneGraph views) support for aiding clone analysis to some extent, but no further support for clone management beyond the detection and visualization of clones.

[4] http://devexpress.com/Products/Visual_Studio_Add-in/Coding_Assistance/

However, beside JClone's inability to detect *Type-3* clones, there is no empirical evaluation of its performance and accuracy in detecting the other two types of clones, that it is described to be able to detect. Moreover, incremental detection and the facility to navigate among the selected code clones are not essential features of JClone, which are indeed necessary towards the usability of a desired clone management system.

Nguyen et al. developed JSync [154] as a plugin to the SVN version control system. Earlier prototypes of JSync appeared as Clever [157] and Cleman [156]. JSync detects clones based on similarities among the feature vectors computed over AST representation of the code fragments. To attain computational efficiency, JSync applies $N(N = 16)$ locality sensitive hashing (LSH) functions to first clusters the code fragments into N buckets. Indeed, the effectiveness of such a clustering depends on the definition of those N individual hash functions. But specifications of those hah functions are not reported, and so no further comments can be made on their effectiveness. Upon having the fragments clustered into N buckets, the fragments in each bucket are pairwise compared to determine if they are clones. Finally, the clone pairs are merged to form clone-groups based on the assumed transitive property among the clones. However, such transitive property may not hold for *Type-3* clones, while it may hold for *Type-1* and *Type-2* clones only. This implies that JSync's approach for clone detection may perform well enough for *Type-1* and *Type-2* clones, but not for *Type-3*. The authors of JSync also noted, "For *Type-3* clones, in JSync, the changes are limited to the minor ones with less than a small pre-defined threshold of added/removed program elements" [154]. Moreover, the accuracy of clone detection is proportional to the value of N, and the increase of N also increases computational overhead. JSync incorporates some features for clone management, which are discussed in Section 4.6.1 and Section 4.8.4.

CPD[5] is a part of Java source code analyzer, PMD. SDD [136] is a clone detection algorithm based on n-neighbour distance, index and inverted index. An implementation of SDD[6] is also freely available as a plugin to Eclipse. Simian[7] is another clone detector available as a plugin to Eclipse. Another Eclipse plugin, CloneDigger[8] applies an approach based on AST, suffix tree, and anti-unifcation for detecting clones in source code written in Java or Python. Tairas and Gray [182] also developed a suffix-tree based clone detector as a plugin for the Microsoft Phoenix framework. Despite the integration with IDEs all these tools (CPD, SDD, Simian, CloneDigger, and the tool of Tairas and Gray [182]) offer no support for clone management except for the detection of *Type-1* and *Type-2* clones only [29].

Another Eclipse plugin, CloneDR[9], is an AST-based clone detector that can de-

[5] http://pmd.sourceforge.net/cpd.html

[6] http://wiki.eclipse.org/index.php/Duplicated_code_detection_tool_(SDD)

[7] http://www.harukizaemon.com/simian

[8] http://clonedigger.sourceforge.net/download.html

[9] http://www.semdesigns.com/Products/Clone/

tect *Type-1* and *Type-2* clones, but it also fails to detect Type-3 clones in many scenarios [168]. Beside clone detection, `CloneDR` offers limited support for clone removal as further discussed in Section 4.8.5. `CeDAR` [183] can incorporate the results from different clone detection tools (e.g., `CCFinder`, `CloneDR`, `DECKARD`, `Simian`, or `SimScan`) and can display properties of the clones in an IDE. `CeDAR` offers no further support for clone management, except that those clone properties may be useful for clone analysis. Moreover, it may suffer from the limitations of the underlying clone detector used internally. Recently, Zibran and Roy [205] developed an Eclipse plugin to facilitate focused search for clones of a selected code fragment. They applied a suffix-tree-based k-difference hybrid approach to detect both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones. They are also extending their tool towards a versatile clone management tool [204].

Table 4 summarizes the the capabilities of the integrated tools in terms of the types of clones they can detect and the features for clone management they offer. However, clone management activities are discussed in more detail in the subsequent sections.

4.5 Clone Documentation

Some clone detectors (e.g., `CCFinder`) record clone information in terms of clone pairs while some other clone detection tools (e.g., `NiCad`) record information in terms of clone-groups consisting of two or more cloned code segments. Typically, clone detectors describe a piece of code fragment (as well as a clone fragment) using the host file's name along with the range of line numbers [164], or the file-name along with character offset and length of the code fragment [83].

Different clone detectors report the results of clone detectors in different formats such as XML, HTML, and plain text. There are variations in the reported information as well. Some clone detectors report clone pairs only, while some other tools report clones in terms of clone-groups. Such variations make it difficult for data exchange between clone detectors, which also adds to the challenges in head-to-head empirical comparison of clone detectors. To minimize the differences in the presentation of clone information, Harder and Göde [72] recently proposed the *Rich Clone Format (RCF)*, an extendible schema based data format for storage, persistence, and exchange of clone data.

Note that, the description of code regions in terms of absolute locations in the host file can be invalidated when changes in the file alters the line numbers, even if the changes are not inside the code/clone regions of interest. To minimize this threat, in `Wrangler`, relative locations of program entities are used instead of absolute locations [139]. The starting line number of each function in its host file is recorded, and with the relative location, every function considered to have started from line 1 at column 1. This was expected to save clone regions from being invalidated due to

Table 4: Summary of clone management support from integrated tools

Tool	Integration with	Clone detection approach	Supported clone types	Support for management
`CloneDetective` [102]	ConQAT	suffix tree	1, 2	detection, visualization
`SimScan`	Eclipse, JBuilder, intelliJ IDEA	AST based	1, 2, 3	detection only
`Simian`	Eclipse	unknown	1, 2	detection only
`DupMan` [63]	Eclipse	uses `SimScan`	1, 2, 3	detection only
`CloneScape` [37]	Eclipse	clip-board operations	copy-pasted	detection only
`CloneBoard` [45]	Eclipse	clip-board operations	copy-pasted	detection, linked editing
`CPC` [197]	Eclipse	clip-board operations	copy-pasted	detection, linked editing
`CnP` [83]	Eclipse	clip-board operations	copy-pasted	detection, consistent renaming
`SHINOBI` [112]	MS Visual Studio	suffix array	1, 2	detection only
`Wrangler` [138]	Eclipse, Emacs	AST, anti-unification	1, 2	detection, tracking, folding
`JClone` [8]	Eclipse	AST based	1, 2	detection, visulaization
`JSync` [154]	SVN	AST, LSH, feature vector	1, 2, 3*	detection, tracking, synchronization, merging, annotation
`CPD`	PMD	fingerprinting	1, 2	detection only
`SDD` [136]	Eclipse	index, n-neighbour	1, 2	detection only
`CloneDR`	Eclipse	AST based	1, 2, 3*	detection, *extract method* refactoring
`CloneDigger`	Eclipse	AST, suffix tree, anti-unification	1, 2	detection only
`CodeRush`	MS Visual Studio	unknown	1, 2, 3*	detection, consolidation
Tool of Tairas and Gray [182]	MS Phoenix framework	suffix tree	1, 2	detection only
Tool of Zibran and Roy [205]	Eclipse	fingerprinting AST, suffix tree	1, 2, 3	detection only

* limited subset of *Type-3* clones

the changes in the pure locations.

Duala-Ekoko and Robillard [49, 51] proposed clone region descriptor (CRD) to describe clone regions within methods in a way that is independent from the exact text of the clone region or its location in a file. The definition CRDs in extended BNF form is as follows:

$$
\begin{aligned}
< \text{CRD} > &::= < file >< class >< \text{CM} > [< method >] \\
< method > &::= < signature >< \text{CM} >< block > * \\
< block > &::= < btype >< anchor >< \text{CM} > \\
< btype > &::= \text{‘for’} \mid \text{‘while’} \mid \text{‘do’} \mid \text{‘if’} \mid \text{‘switch’} \mid \text{‘try’} \mid \text{‘catch’}
\end{aligned}
$$

Thus, CRD describes a code/clone region in terms of the distinguishing descriptions of the enclosing file, class, method, block, and a corroboration metric (CM). The corroboration metric used is simply the addition of cyclomatic complexity and fan-out of the block. The $< anchor >$ for a block is a distinguishing string description of for the block, and the derivation of the description is dependent on the type of the block. The termination statement for loops, the branching predicate for if blocks, and the switch expression for the switch statements are used as $< anchor >$. For the try blocks, CRD uses the list of exception types caught in catch clauses associated with the block. For catch blocks, simply the type of the exception caught is used as the $< anchor >$. Using the CRD scheme, Duala-Ekoko and Robillard developed `CloneTracker` [50] for tracking clones during the evolution of the software code base. `CloneTracker` uses `SimScan` as the underlying clone detector, allows the developer to select individual clone-groups, and supports simultaneous modifications of clone regions.

However, such a scheme like CRD [49] has a number of limitations. First, small changes in the code corresponding to the $< anchor >$ (e.g., termination condition of loop, branching predicate of conditional statements) will invalidate the CRD. Second, the scheme is vulnerable to nesting levels, and thus a simple addition or removal of nesting level will invalidate the CRD. Third, the association of ‘else’ blocks with the closest ‘if’ block prevents the CRD scheme to differentiate between the two types of blocks. Most importantly, the use of the CRD scheme did not save `CloneTracker` [50] from re-invoking the underlying clone detector to identify possible changes in the clones, though the computational expense of re-detection was indicated as one of the motivations behind the design of CRD.

Markers based tagging support in IDEs like Eclipse can be used for clone documentation. Such tagging of clones can provide built-in support for accommodating changes in the source files [37].

4.6 Clone Tracking

During the development of an evolving software system frequent changes take place in the code base. Such changes may introduce new code segments that might form new clones. Moreover, changes in source files may invalidate the clone regions necessitating corresponding updates in the recording of clone information. Such updates can be done in two ways: Re-detection and incremental detection.

Re-detection: The detection of clones from the entire system may be invoked every time the code changes. This approach may incur too much overhead as the detection of code clones in a fairly large system can be computationally expensive. Hence, the approach is unlikely to be suitable for proactive clone management.

Incremental Detection: A better approach is incremental detection, where only the source code in the modified portion of the code base is examined for any clones and the outcome is accumulated with the previously preserved clone detection results.

4.6.1 Incremental Clone Detection

Not many attempts made towards incremental clone detection. The first attempt was made by Göde and Koschke [64, 67]. They proposed a suffix tree based algorithm `iClone` [64, 67] for incremental detection of clones in subsequent versions of a given system. As input, `iClone` needs n revisions of a program that is to be analyzed. In addition to the source code for each revision, `iClones` also needs a separate file as input that summarizes in `Subversion`[10]'s format, all the changes in the source files for every two consecutive revisions. Each source file is represented as a sequence of tokens extracted using lexical analysis, and tokens are stored in a token table. For the detection of clone in the initial revision, a generalized suffix tree (GST) is constructed for a large sequence of tokens obtained by concatenating token sequences from all the source files. The paths from the root all the non-leaf internal nodes point to clones in the code [205]. The GST is preserved and updated during the detection of clones in the remaining revisions. For the detection of clones in revision i, only the files that are changed, added or deleted between revision $i-1$ and revision i are examined, and in accordance, the GST is updated.

`iClone`'s approach for clone tracking appears to be memory intensive [67, 85], due to the use and maintenance of a large GST constructed from all tokens of the entire program. Moreover, the tokens of each source file is stored in separate token table, and thus a large number of token tables also need to be stored, which may also consume a significant amount of memory. Indeed, the suffix tree based clone detection technique of `iClone` can detect *Type-1* and *Type-2* clones only, but not *Type-3* [67, 205]. Besides, the need for the input as a file summarizing changes between subsequent

[10] `http://subversion.tigris.org/`

versions further limits the applicability of the technique in the implementation of a clone management system having *decentralized* architecture. However, the approach can be a suitable for clone tracking in a *centralized* clone management system, where the source code is kept in a central `Subversion` repository and the code change information can easily be made available.

Hummel et al. [85] proposed an index-based incremental clone detection approach. They demonstrated their technique as a pipeline of three phases: preprocessing, detection, and post-processing. The preprocessing and post-processing components were reused from `ConQAT`. The detection component introduces *clone index*, the central data structure of their approach. The *clone index* is a list of tuples *(file, statement index, sequence hash, info)*, where, *file* is the name of the source file, *statement index* is the position in the sequence of normalized statements for the *file*, *sequence hash* is a MD5 hash code computed over the chunk of n normalized statements from the *statement index*, and *info* contains additional data such as the start and end lines of the chunk of consecutive statements. The heart of the technique lies in the *sequence hash*, tuples with the same sequence hash indicate clones containing at least n statements. Consecutive such clones are further merged to report only maximal clones.

Similar to `iClone`, the technique of Hummel et al. is also limited in detecting *Type-1* and *Type-2* clones, but not *Type-3*. The first step of the detection algorithm is to create, for each file, a list of duplicated chunks. For a large software system, preserving such lists can cause significant memory consumption and updating those with the changes in the code base can incur significant computational overhead. The MD5 hashing algorithm, specially for small n, might produce same hash value for different chunks of statement, and thus may cause false positives in the clone detection [85]. The technique can also appear inefficient, when the list of tuples are not maintained as a sorted list, while such maintenance may frequently invoke sorting overhead whenever the code changes.

Higo et al. [81] also proposed a PDG-based incremental clone detection technique, where PDGs are generated from the analysis of control and data dependencies in the program code. The PDGs are preserved in the database, and clone detection is performed by approximate comparison of PDGs. The PDGs in the database are kept in sync with the evolving source code by examining only the updated source files. As mentioned in Section 3.5.1, PDG based techniques are computationally expensive and they often report *non-contiguous* clones that may not be perceived as clones by a human evaluator.

Li and Thompson [139] enhanced the clone detection technique of `Wrangler` by introducing incremental detection. The initial clone detection is performed in two steps. First, the source code is normalized and parsed to produced AST. The AST is then annotated and serialized. Then a suffix tree based approach is applied to the

Table 5: Summary of tool support for incremental clone detection

Tool	Technique	Supported clone types	Integration with IDE/VCS
`iClone` [67]	Preservation of suffix tree	1, 2	Separate tool
Tool of Higo et al. [81]	Preservation of PDGs	non-contiguous clones	Separate tool
Tool of Hummel et al. [85]	MD5 Hashing and indexing	1, 2	Integrated with ConQAT
`Wrangler` [139]	AST serialization suffix-tree, anti-unification	1, 2	Integrated with Eclipse, Emacs
`JSync` [154, 155]	AST, LSH, feature vector	1, 2, 3*	Integrated with subversion

* limited subset of *Type-3* clones

serialized AST for detecting the initial clone candidates. The second step applies *anti-unification* technique to get rid of the false positives. The annotated AST is preserved, which is updated whenever changes take place in the source code.

The approach maintains a table, where the annotated AST representation of each expression statement is preserved. The storage and update of the table might cause significant memory consumption and computational cost. Moreover, the approach is limited in the detection of only *Type-1* and *Type-2* clones in functional programs written in Erlang.

The clone tracking approach of `JSync` [154, 155] appears to be computationally elegant. `JSync` preserves the clone-groups and N buckets obtained from the initial clone detection. Since `JSync` is implemented as a plugin to SVN, the change information of the source files are readily available, and based on that information `JSync` can determine the fragments modified, added to, or deleted from the code repository. `JSync` then removes from the clone-groups those fragments that were changed or deleted. Then the LSH technique is applied to the newly added and modified code fragments to place them in the buckets. Then the fragments in each bucket are compared pair-wise to update the clone-groups. Thus, the clone detection technique of `JSync` appears to be inherently incremental and consequently computationally efficient for tracking clones.

In Table 5, we summarize the techniques and tools proposed for incremental clone detection. As can be noted, all the tools have vivid limitations in dealing with *Type-3* clones, and the storage of a high volume of data has been a common issue with all the techniques. Further research in the area my inform techniques for integrated incremental detection of clones including *Type-3* with better storage efficiency.

4.7 Clone Annotation

The developers often deliberately create clones, for example, to enable independent evolution of similar implementations. During the clone management process, the developer may not want to refactor/remove those clones, and may want to mark those to indicate such decisions so that they won't have to encounter those same set of clones over and over. Moreover, the decision needs to be documented and shared among different programmers, and there should be facility for the developers to review those clones later time, in case they want to re-evaluate their management decision. To the best of our knowledge, such a feature is found only in `JSync` [154], which allows the developer to annotate pairs of clones for avoiding future encounters.

4.8 Techniques for Reengineering/Refactoring of Clones

The investigations of opportunities for clone based reengineering and refactoring of clones for their removal have suggested techniques such as generics, design patterns, software refactoring patterns, and synchronized modifications of code clones.

4.8.1 Generics and Templates

Basit el al. [24] investigated the potential of generics in removing code clones. They carried out two case studies on the *Java Buffer Library* and the *C++ Standard Template Library (STL)*. The *Java Buffer Library* was found to have 68% redundant code, and using generics they were able to remove only 40% of them. Though, they performed little better for the *C++ STL*, they concluded that the constraints of language constructs limit the applicability of generics in clone removal. They further hypothesized that the meta level parameterizations might perform better as they are relatively less restrictive than generics or templates.

The hypothesis on the potential of meta level parameterizations was addressed by Jarzabek and Li [90] in a later study. They also used the *Java Buffer Library* for their case study. They applied a generative programming technique using *XVCL (XML-based Variant Configuration Language)*[11] to represent similar (but may not be identical) classes and methods in generic and adaptable form. Using the technique they were able to eliminate 68% of the code from the original *Java Buffer Library*.

4.8.2 Design Level Approaches

Design Patterns: Balazinska et al. [17] attempted to replace code clones by applying the *strategy design pattern* for partial redesign of Java systems. The idea was to factorize commonalities in the cloned methods and parameterize their differences

[11] `http://xvcl.comp.nus.edu.sg/`

to preserve the original behaviours, and then weave them through the *strategy design pattern*. Their approach was realized in a tool named `CloRT`, and the reengineering technique was applied to the source code of JDK 1.1.7 for empirical evaluation. However, the *synchronized* (thread safe) methods were kept out of the study. The reengineering process merged the source of 28 methods, but created 84 new methods, and thus actually increased the line of code (LOC). Indeed, LOC can be a fair estimation for the size of the code base, but not for design quality. Use of a design quality metric suite could have been used to reflect the actual impact of the reengineering technique on the quality of of the code. The use of other design patterns, such as the *factory pattern* may also produce similar result. Further investigation is needed towards this possibility.

Traits: Traits [176] are a modularity mechanism that complements inheritance to facilitate an orthogonal means of sharing functionality in object oriented classes. Traits can be considered as a language extension, and essentially, a trait is a set of pure methods (methods that do not directly refer to any instance variable), which can be used in a class or another trait simply by name. Traits can also be a potential mechanism for removing duplicated code, specially when it becomes difficult due to restrictions from inheritance hierarchy. Murphy-Hill et al. [152] applied the traits mechanism to remove duplicated code from the *java.io* library. Using 14 traits, they were able to get rid of 30 duplicated methods by refactoring 12 classes.

Aspects: Aspects [115] and aspect-oriented techniques (AOT) support the modularization of features that cross-cut the class hierarchy. AOT can be promising in improving modularization, which consequently may reduce code clones. The behaviours that Murphy-Hill et al. [152] parcelled up into traits could also be put into different aspects, and introduced into the target by a mechanism like intra-class declarations of AspectJ [152]. On the contrary, Jarzabek and Li [90] argues that due to the lack of parameterization mechanism and constrained composition rules, AOT in its pure form is not meant for elimination of redundancies. More investigation in this regard is necessary, which might reveal interesting results and opportunities.

4.8.3 Synchronized Modification

Simultaneous editing has been a popular approach for applying the same editing operations to multiple segments of identical text.

An early work [150] on simultaneous editing incorporated the feature in a text processing system called `LAPIS` [149]. `LAPIS` offers a library of built-in parsers and patterns for various kinds of text structure, including HTML and Java source code. The user is enabled to select multiple regions of text (manually or by using patterns) and perform simultaneous editing.

Toomim et al. [188] proposed to apply simultaneous editing to simultaneously edit duplicated code. They called such editing as "linked editing", and to support this inside editor, they developed `CodeLink` as an extension to the XEmacs editor. The "linked editing" of `CodeLink` is functionally same as the "simultaneous editing" of `LAPIS`. However, `CodeLink` saves the user from explicitly select multiple regions for edit as to be done with `LAPIS`. Instead, in `CodeLink` the user manually selects two code segments to link them. Then `CodeLink` applies a LCS (longest common subsequence) algorithm to map between the differences and similarities in the code segments, and enables simultaneous editing in the duplicated regions. However, due their use of the dynamic programming version of LCS algorithm, their approach appears memory intensive and computationally expensive. For k clones, each of n tokens, the dynamic programming implementation of LCS algorithm runs in $O(n^k)$ time. Moreover, `CodeLink` "does not always report the most intuitive set of differences between any two code fragments" [188]. The support for linked/simultaneous editing is also available in tools such as `CPC` [197], `CloneBoard` [45].

4.8.4 Consistent Renaming

Programmers often perform modifications after copy-pasting a code fragment. Such modification typically include renaming of identifiers according to the new context of the cloned code. IDEs like Eclipse provide necessary support for consistently renaming an identifier and all its references within scope. Jablonski and Hou developed `CReN` [86] as a plugin to Eclipse that can check for any inconsistencies in the renaming of identifiers within a code fragment and suggest modifications for making the renaming consistent. They further extended `CReN` and developed `LexId` [87], which supports consistent modification of the same parts of different identifiers in a code segment. However, the consistent renaming support from these tools are limited to within a single code fragment, unlike the linked/simultaneous editing between clone pairs [83]. Jacob et al. developed `CSeR` [88] by extending the Java editor of Eclipse to visualize the similarities and differences while a programmer edits a copy-pasted code. Hou et al. combined `CReN` and `CSeR` into a single toolkit named `CnP` [83], which they are developing towards support for proactive clone management based on copy-paste clipboard activities.

Since `JSync` [154] is developed as a plugin to the SVN version control system, it can exploit the change information between versions of Java source files to determine whether any changes occurred in cloned code regions. For such a clone pair it maps between the nodes of ASTs of the corresponding code using a *treed* algorithm, detects inconsistencies in the identifier renaming, and suggests for consistently renaming them. This feature of `JSync` is very similar to that of the consistent renaming support of `CReN` [86]. `JSync` also supports *clone synchronization* between clone pairs when one of the clones changed between version while the other remained unchanged.

Clone synchronization of JSync simply accommodates the changes from the modified code fragments into the unchanged code. In case both the clone pairs changed between versions, JSync suggests *clone merging*. For clone merging, JSync suggests to accommodate the changes from both the clone fragments to each other, and any conflict is simply left to the user's resolution. Moreover, the clone synchronization support of JSync is limited to simple changes in the identifiers, control structures, literals, and method calls. However, JSync also allows annotating clone pairs in case the developer wants to retain the inconsistencies. Through an empirical evaluation, Nguyen et al. [154] reported that JSync can attain 83% precision in recommending change propagation between clone pairs.

4.8.5 Refactoring Patterns

Fowler in his book [58], presented 72 patterns for refactoring source code in general for the removal of code smells. Over time the number of refactoring patterns has increased to 93, and a *refactoring catalog*[12] is maintained that lists and describe them all. Among those general software refactoring patterns [58], the following patterns are found to be suitable for clone refactoring, as suggested by earlier research [75, 76, 104, 119, 135, 177, 199, 202, 203]. Detail about these refactoring patterns can be found at the *refactoring catalog* and elsewhere [58].

- **Extract method** extracts a block of code as a new method, and replaces that block by a call to the newly introduced method. EM may cause splitting of a method into pieces. For code clone refactoring, similar blocks of code can be replaced by calls to an extracted generalized method.
- **Move method** relocates a method from one class to another class as appropriate. This can be used to merge identical methods [135].
- **Pull-up method** removes similar methods found in several classes by introducing generalized method in their common superclass.
- **Extract superclass** introduces a new common superclass for two or more classes having similar methods, and then applies *Pull-up method*. This may be necessary when those classes do not have a common superclass.
- **Extract utility-class** is applicable in situations, where similar functions are found in different classes, but those classes do not conceptually fit to undergo a common superclass. A new class is introduced that accommodates a method generalizing the similar methods that need to be removed from those classes.

[12]Catalog of OO refactoring patterns: http://refactoring.com/catalog/

- **Rename refactor** is simply altering the names of variables, methods, classes and so on. The removal/generalization of near-miss (similar, but not exact duplicate) clones will require fine grained modifications, such as, changes in identifier names, method signatures and type declarations to reduce the textual differences between clone pairs.

Besides these prominent refactoring patterns, other low level refactoring operations such as, *identifier renaming*, *method parameter re-ordering*, *changes in type declarations*, *splitting of loops*, *substitution of conditionals*, *loops*, *algorithms*, and *relocation of method or field* may be necessary to produce generalized blocks of code from near-miss (similar, but not exact duplicate) clones [203]. Kerievsky [114] proposed the *chained constructor* refactoring pattern[13], which can also eliminate duplicated code from the constructors of the same class [153]. Other refactoring patterns that can be found in the literature are some sort of variants or compositions of the aforementioned object-oriented refactoring (OOR) patterns. Other than the OOR patterns, Schulze et al. [178] proposed three aspect oriented patterns described as *extract feature into aspect, extract fragment into advice*, and *move method from class to interface*.

Tool Support for Refactoring Patterns: Though a variety of potential refactoring patterns have been identified for code clone refactoring, tool support for the automated or semi-automated application of those refactoring operations has been quite limited. A number of semi-automated analytical techniques have been proposed for finding candidate clones that are easier and suitable for refactoring. Such techniques are described in Section 4.9.2. In this section, we focus on the techniques realized in tools for the modification of the candidate clones in actual refactoring operations.

Limited support for the *rename refactoring, extract method, extract superclass* and *pull-up method* refactoring can be available from IDEs like Eclipse, when the necessary program element are manually identified by human efforts. To aid refactoring of clones in object-oriented code written in C++, Fanta and Rajlich [56] developed tool support for five high level restructuring features namely, *function insertion, function expulsion, function encapsulation, renaming*, and *argument reordering*. However, the implementation of those features are very limited in operational scope. For example, the inserted function cannot be a member of any class, cannot be overloaded, cannot be a template function, and cannot be called using pointer.

`Wrangler` [138] supports clone removal by *folding* expressions against a function definition in Erlang programs. Folding searches the program for instances of the right-hand side of the selected function clause, and under the user's control, it replaces them

[13]Catalog of 27 refactoring patterns from J. Kerievsky's book: `http://industriallogic.com/xp/refactoring/catalog.html`

with applications of the function to the actual parameters. The *folding* operation in functional languages is similar to the *extract method* refactoring in imperative languages.

`CloneDR` provides facility to automatically or interactively perform *extract method* refactoring by identifying identical blocks of code and producing a parameterized function/method that generalizes the blocks. Recall that `CloneDR` has limitation in detecting *Type-3* clones [168], and the application of extract method refactoring may be simpler for *Type-1* and *Type-2* clones, which is likely to be much more cumbersome for *Type-3* clones due to addition or deletion of statements at arbitrary locations of the clone fragments.

4.9 Analysis and Identification of Clones for Refactoring

For the purpose of finding and characterizing code clones suitable for refactoring, reengineering, or removal, in depth analysis of the various properties of the clones and their context is required. Clone visualization has been proven to be effective in aiding such analysis. Therefore, we first discuss the tools and techniques for code clone visualization, and then we present the findings from analysis of code clones in search for clone based reengineering opportunities.

4.9.1 Visualization of Distribution and Properties of Clones

A major challenge in identifying useful cloning information is to handle the large volume of textual data returned by the clone detectors. To mitigate the problem, a number of visualization techniques, filtering mechanisms and support environments are proposed in the literature. Jiang et al. [94] categorized the proposed clone presentation techniques based on two dimensions. The first dimension refers to the level at which the entities are visualized (such as at the code segment level or file level or subsystem level). The second dimension refers to the type of clone relation addressed by the presentation (whether clones are showed at the clone pair level or grouped into clone classes or super clones).

Johnson [97] used the popular Hasse diagram to represent the textual similarity between the files. Later, he also proposed hyper-linked web pages to explore the files and clone classes [98]. Cordy et al. [41] used HTML for interactive presentation of clones where overview of the clone classes is presented in a web page with hyperlinks and users can browse the details of each clone class by clicking on those links. Although such representations offer quick navigation, they cannot reveal the high level cloning relations. A set of polymetric views [162] were also proposed in the literature that permit encoding of a number of code clone metrics to the visual elements. Among various visualizations, scatter plot is the most popular and capable of visualizing inter-system and intra-system cloning [40, 144]. However, the size of

the scatter plot depends on the size of input rather than the amount of cloning. Thus, using a scatter plot for visualizing cloning relation of a large software system may become challenging due to the large size of the plot. Moreover, non-contiguous sections that contain the same clone cannot be group together. To overcome this, Tairas et al. [185, 186] proposed a graphical view of clones (also known as Visualizer view) that represents each source file as a bar and clones within the files are represented with stripes. Clones belong to the same class are encoded with the same color.

Jiang et al. [93] extended the idea of cohesion and coupling to code clones and proposed a visualization that uses shape and color to encode the metric values. They also developed a framework [93] for large scale clone analysis and proposed another visualization, called a clone system hierarchical graph that shows the distribution of clones in different parts of a system. Fukushima et al. [59] developed another visualization using graph drawing technique to identify diffused (scattered) clones. Here, nodes represent the clones. Those nodes that are located in the same file are connected with edges to form a clone set cluster. Nodes that connect different clone set clusters are called diffused clones (have cloning relation in different files implementing different functions).

`Gemini` [192] is an example of a clone support environment that uses CCFinder for clone detection and can visualize cloning relation using scatter plots and metric graphs. Kapser and Godfrey developed `CLICS` [108, 111], another tool for clone analysis. `CLICS` can categorize clones based on their previously developed clone taxonomy [107] and support query based filtering. However, it is limited to only C/C++ and Java source code. Tairas et al. [186] developed an Eclipse plug-in that works with `CloneDR`, a clone detection tool and implements the visualizer view along with general information and detected clones list views. Third in this group is the `Clone Visualizer` [200], an eclipse plugin that works with `Clone Miner`, a clone detection tool. In addition to supporting clone visualization through stacked bar chart and line graph, it supports query based filtering. The recent addition to this group is `CYCLONE`[14] [72]. It supports single and multi-version program analysis and uses RCF (Rich Clone Format) [72] file as an input. RCF is a data exchange format capable of storing clone detection results. A separate viewer application named `RCFVIEWER`[15] is also developed for the visualization of clone information stored in RCF format.

Table 6 summarizes the various clone visualization techniques realized in different tools. As can be noted, all the visualization techniques focus on visualization of clone pairs or clone-groups with respect to their dispersion in the file-system hierarchy. However, from the perspective of clone removal or refactoring, the visualization of the clones with respect to the inheritance hierarchy can offer useful insights, and future work in clone visualization should address this possibility.

[14] `http://softwareclones.org/cyclone.php`
[15] `http://www.softwareclones.org/`

Table 6: Summary of clone visualization techniques (extended from Jiang et al. [94])

Visualization Technique	Granularity	Clone Relation
Duplication Aggregation Tree Map [5, 6, 162]	File, Subsystem	Clone Group
Scatter/Dot Plot [5, 6, 73, 162, 192]	File, Subsystem	Clone Pair
System Model View [162]	File, Subsystem	Clone Pair
Clone System Hierarchy Graph [93]	File, Subsystem	Clone Pair Clone Group
Hasse Diagram [97]	File	Clone group
Clone Group Family Enumeration [162]	File	Clone Group
Duplication Web [162]	File	Clone Pair
Dependency Graph [5, 6, 108]	Subsystem	Clone Pair
Clone Coupling and Cohesion [94]	Subsystem	Super Clone
Metric Graph [192]	Code Segment	Clone Group
Clone Cluster View [59]	Code Segment	Clone Group
Hyper-Linked Web Page [41, 98]	Code Segment	Clone Group
Clone Visualizer View [185, 186]	Code Segment, File	Clone Group
Stacked Bar Chart [200]	Code Segment, File	Clone Group
Line Graph [200]	Code Segment, File	Clone Group

4.9.2 Analysis to Find Clone Based Reengineering Opportunity

Early works [10, 25] on exploiting code duplication, proposed macro extraction from source code (C and C++). While those approaches pioneered the clone refactoring research, they were focused on simple refactoring in procedural code and did not tackle the issues that might raise in refactoring clones in object-oriented context.

Ducasse et al. [53] proposed to use clone detection tools for guiding the refactoring of code clones. For the detection of code clones in SmallTalk code, they used `DUPLOC`, a text-based clone detector that performs basic string matching over the lines of source code. From the analysis of the identified duplicated code, they found two cases where clone refactoring operations can be applied. Those two cases were simply two variations of the well-known *extract method* [58] refactoring pattern.

For the identification of code clones as suitable candidates for refactoring, Balazinska et al. [15] proposed advanced clone analysis based on similarities and difference in the code clones. They also suggested a number of measurements for context analysis, which capture the relationships between a class and its member methods, between a method and its member variables, as well as the caller-callee relationships among the methods. Though their work focused on the refactoring of code clones in object-oriented systems, they completely ignored the horizontal and vertical dependencies due to the inheritance hierarchy. Based on a similar analysis, Zibran and Roy proposed a code clone refactoring effort model [203], which takes into account all the

relationships including the inheritance hierarchy.

Ueda et al. developed `Gemini` [191, 192], a graphical tool to aid clone analysis for corrective maintenance of code clones. `Gemini` works on top of the `CCFinder` clone detector, and facilitates the visualization of clones using Scatter Plot. It also computes a number of metrics (i.e., RAD, LEN, POP, DFL, RNR) to capture various properties of the detected clones [77]. A metric-graph view enables the user to set upper and lower bounds on each metric to filter out those clones that falls beyond the boundaries, per se, the user's interest [38, 76]. Since `CCFinder` cannot detect *Type-3* clones, the functionality of `Gemini` stayed limited in *Type-1* and *Type-2* clones only. However, they demonstrated that the Scatter Plot of `Gemini` enabled the users' subjective perception to visualize *Type-3* (gapped) clones, without actually detecting them automatically [193].

As `CCFinder` detects clones that can be any arbitrary lines of code in the code base, those often lack the contextual cohesion necessary for effective refactoring. To minimize this issue with `Gemini`, Higo et al. added to `Gemini` a feature called `CCShaper` [76, 79] that applies a parser-based technique (using JavaCC) to extract the block level clones from `CCFinder`'s clone detection result. The objective was to extract block clones which might be easily merged by applying the *extract method* refactoring pattern. A tool named `ARIES` [75] was also published, which combined the metric view of `Gemini` and the clone filtering feature of `CCShaper` while introduced two additional clone metrics (i.e., NRV and DCH) beyond those previously existed in `Gemini`.

Method/function level code clones belonging to different clone-groups, at times, may have dependency in terms of caller-callee or data sharing (reference or assignment) relationship. Yoshida et al. [199] called such clones as "chained clones", and argued that all of such clones should be refactored at once. Thus, they extended `ARIES` with a PDG based technique to *identify* such chained clones from the initially detected clones. Indeed, due to the use of `CCFinder` as the backend clone detector, their automated technique can handle only *Type-1* and *Type-2* clones, but not *Type-3*.

Tairas and Gray [184], through an empirical study on two open source Java systems (JBoss and ArgoUML), reported that in some cases clone refactoring was partially performed on only part of the clones, and in most cases those refactored clones were still further refactorable. However, they focused to investigate only the occurrences of refactorings composed of the *extract method* refactoring pattern. Indeed, there are other types of refactorings including small modifications (e.g., identifier renaming, addition/deletion of statement) in the code fragments, which might have caused partial changes in the clones, as they found. Most importantly, they observed changes by comparing clones between pairs of consecutive releases, and simply *assumed* that those changes occurred due to refactoring. However, such an assumption may not hold true, as those small changes may be caused by many reasons such as

corrective or adaptive maintenance, or to change functionalities of those code fragments. Moreover, "it is unlikely that developers of JBoss and ArgoUML used a clone detection tool to identify clones" [184], or used any tool support for clone refactoring. This might be a plausible reason why there were small changes in the clones that still remained refactorable from the perspective of clone removal. This rather emphasizes the need for effective clone management tool support for software maintenance.

Higo et al. developed `Libra` [80], a tool on top of the `CCFinder` clone detector to facilitate the detection of clones of only a given code fragment, instead of detecting all clones from the entire given code base. They argued that such a clone detection may be useful to find potential clones where simultaneous modifications can be applied. `Libra` was not integrated with any IDE. Moreover, since it internally invokes `CCFinder`, it cannot handle *Tyep-3* clones. Lee at al. [134] used an algorithm based on feature-vector computation over AST and finds the k most similar clones of a given code segment. But, their tool was not reported to have integration with IDE. Zibran and Roy [205] developed a similar tool which is integrated with the Eclipse IDE, and can detect all the three types (*Type-1, Type-2*, and *Type-3*) of clones.

Clone Categorization Based On Reengineering Opportunity: Analysis of diverse characteristics and properties of code clones in quest of reengineering opportinuties led to different taxonomies of code clones, and refactoring strategies for different categories of code clones.

From a manual analysis of 800 function/method level clones over six different open-source Java systems, Balazinska et al. [16] proposed a taxonomy of function clones (Table 7) based on the differences and similarities in the program elements.

Taking into account the location of clones in the inheritance hierarchy, Koni-N'Sapu [121] proposed a clone taxonomy (Table 8) and a set of object-oriented refactoring patterns for refactoring each categories of code clones.

Schulze et al. [178] argued that *aspect-oriented refactoring (AOR)* can be more appropriate than *object-oriented refactoring (OOR)* in certain scenarios. They proposed a code clone classification scheme to support the decision whether to use OOR or AOR for clone removal. The classification is based on two dimensions: type and location. Type specifies the kind of statements (i.e., conditionals, loops, functions, and other) that constitute the clones. The location dimension reflects the dispersion of the clones in terms of their host source files in the file-system hierarchy. To capture such regional information for a clone-group $G = \{C_1, C_2, C_3, \ldots, C_n\}$, they proposed

Table 7: Balazinska et al. [16] Taxonomy: categories of function/method level clones based on (dis)similarity caused by different types of differences

#	Categories of clones	Description of (dis)similarity among methods/functions
1	Identical	strictly identical functions/methods
2	Superficial changes	differences that do not cause difference in behaviour
3	Called methods	differences only in some method calls
4	Global variables	single-token differences corresponding to non-local variables or constants
5	Return types	single-token difference corresponding to the return type
6	Parameter types	single-token differences corresponding to parameter types
7	Local variables	single-token differences corresponding to the types of local variables
8	Constants	single-token differences correspond to constants hard-coded in the methods
9	Type usage	single-token differences corresponding to types explicitly manipulated in expressions line "instanceof" or "typecast"
10	Interface changes	single-token differences corresponding to called methods and/or global variables and/or parameters types and/or return type
11	Implementation changes	single-token differences corresponding to types of local variables and/or constants used and/or types explicitly manipulated
12	interface and implementation changes	single- token differences corresponding to any difference used in the definition of the previous categories
13	One long difference	long difference in one entity (e.g., expression, statement)
14	Two long differences	long differences in two entities (e.g., expression, statement)
15	Several long differences	long differences in three or more entities
16	One long difference, interface, and implementation	combination of differences as of categories 12 and 13
17	Two long differences, interface, and implementation	combination of differences as of categories 12 and 14
18	Several long differences, interface, and implementation	combination of differences as of categories 12 and 15

Table 8: Koni-N'Sapu [121] Taxonomy: applicability of different refactoring patterns on different categories of clones

		Refactoring patterns							
		Extract method	Insert method call	Insert super call	Parameterization	Pull-up method	Form template method	Push-down method	Extract superclass
Categories of clones	In the Same Method	√	√		√		√		
	In the Same Class	√	√		√		√		
	With a Sibling Class	√			√	√	√		√
	With the Superclass			√	√	√	√	√	
	With an Ancestor	√			√	√	√		
	With a First Cousin	√			√	√	√		√
	In Common Hierarchy	√			√	√	√		√
	In Unrelated Classes								

a metric $DIST_G$, which is computed as follows:

$$
\begin{aligned}
DIST_G &= scale \times (W_{vert} \times vGap_G + W_{hor} \times hGap_G) \\
scale &= \frac{1}{\max\limits_{C_i \in G}\{depth(C_i)\}} \\
vGap_G &= \max_{C_i, C_j \in G}\{vGap(C_i, C_j)\} \\
hGap_G &= \max_{C_i, C_j \in G}\{hGap(C_i, C_j)\} \\
vGap(C_i, C_j) &= |depth(C_i) depth(C_j)|, \quad C_i, C_j \in G \\
hGap(C_i, C_j) &= \max\{vGap(C_i, C_R), vGap(C_j, C_R)\}, \quad C_i, C_j, C_R \in G
\end{aligned}
$$

Here, $depth(C_i)$ refers to the nesting distance of the source file containing clone C_i from the root of the project's root. C_R denotes the closest enclosing directory/file that hosts both clones C_i and C_j.

Based on the classification with respect to the two dimensions, Schulze et al [178] suggested feasibility of three OOR and three AOR operations as presented in Table 9. However, any empirical evaluation of the effectiveness of their clone classification scheme or the clone removal procedure is not available. A similar workflow for the

Table 9: Schulze et al. [178] Taxonomy: applicability of object-oriented (first three) and aspect-oriented (last three) refactoring patterns to refactor categories of clones

	Conditional/Loop					Function				
	0.0	< 0.1	< 0.3	< 0.6	< 1.0	0.0	< 0.1	< 0.3	< 0.6	< 1.0
Extract method	++	++	+	−	−−	+	−	−	−−	−−
Pull-up/move method	−−	++	+	−	−−	−−	++	+	−	−−
Form template method	−−	−−	−−	−−	−−	++	++	+	+	+
Extract feature into aspect	−−	−−	+	++	++	−−	−−	+	++	++
Extract fragment into advice	−−	−−	+	+	+	−−	−−	+	+	+
Move method from class to interface	−−	−−	+	+	+	−−	−−	+	+	+

Here, ++ means approved w/o constraints, + means approved with constraints
− means in exceptional cases, −− means not suitable/applicable

detection and refactoring of code clones was proposed by Kodhai et al. [119], the implementation of which is not available, let alone any empirical evaluation.

Torres [189] argued that classification of concepts containing duplicated code can provide hints about which refactoring can be suitable. The applied a concept-lattice based data mining approach to derive four categories of concepts containing duplicated code and suggested refactoring patterns suitable for refactoring clones in each of those categories (Table 10).

Taking into account both the locations of clones in the file-system hierarchy and (dis)similarities in the functionalities, Kapser and Godfrey [107] proposed a taxonomy (Figure 7), which is often considered as the most extensive clone taxonomy to date [125]. Despite the number of proposed clone taxonomies for reengineering opportunities, the state of the art still demands more investigation in this regard suggesting open questions [125], for instance, can other properties such as cost, benefit, risk of refactoring be incorporated in a taxonomy?

4.10 Cost-benefit Analysis and Scheduling of Refactoring

Not much research has been done towards the cost-benefit analysis of code clone refactoring and their scheduling. Bouktif et al. [30] first proposed a simple effort model for the refactoring of code clones in procedural code base. Their model simply

Table 10: Torres [189] Taxonomy: clone categorization based on concept location

Cat.	Description	Refactoring Pattern
1	concepts which have all the similar parse-tree expressions in different elements or methods, and belong to the same class	Extract method
2	concepts having similar methods, with the same name, belonging to different classes but in the same hierarchy, and that does not exist in the respective superclass	Pull-up method
3	concepts having similar methods, with the same name, belonging to different classes but in the same hierarchy, and the respective superclass contains a method with the same signature	Extract class in hierarchy
4	concepts that have similar methods, but belong to two or more different hierarchies	Extract crosscutting class

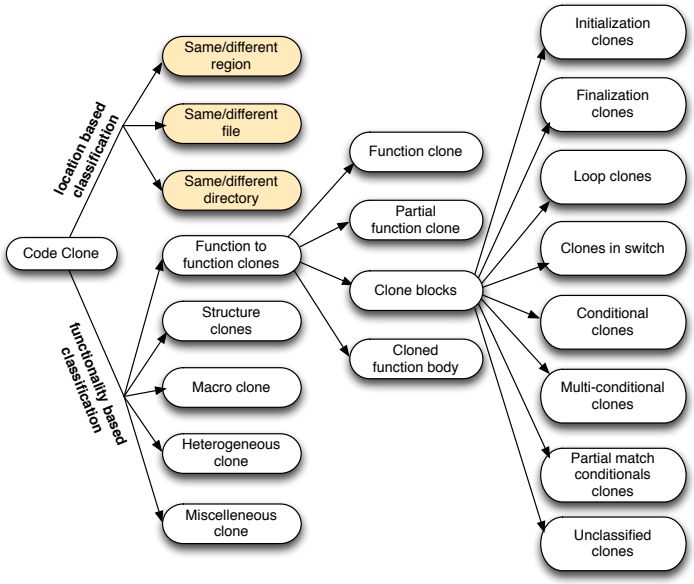

Figure 7: Kapser and Godfrey [107] Taxonomy: clone categorization based on location and functionality

estimates the effort needed to perform a refactoring operation by taking into account the size of the clones in terms of lines of code, caller-callee relationship among the functions, and the number of tokens to be modified. They formulated code clone refactoring scheduling as a constrained Knapsack problem and applied a metaheuristic genetic algorithm (GA) to obtain an optimal solution. Their formulation of the problem had three objective functions: one for maximizing the lack of cohesion in software modularity, another for minimizing coupling, and the other is for increased satisfaction of priorities. However, they ignored all the constraints that might exist among the clone-groups and the refactoring operations. Priorities were given to refactoring of the the old Kernighan and Ritchie style code, of large functions, and of functions with high cyclomatic complexity. The required refactoring effort must not exceed the maximum amount of effort provided by the available resources for the refactoring activities. The efforts needed to refactor near-miss clones were assumed to be thrice the effort required to refactor exact (*Type-1*) clones. However, such an assumption may not hold in practice, due to the fact that clone refactoring effort is much dependent on the context, the number of members in the clone-group, their dispersion in the code base, and so on.

Liu et al. [141] used a heuristic algorithm to schedule refactoring of code bad smells in general. Their work took into account the conflict and sequential dependencies among the refactoring activities, but missed the constraints of mutual inclusion and refactoring efforts. The objective of their scheduler was to maximize a number of code quality attributes while satisfying the constraints. For quality assessment, they used the QMOOD [18] design quality metric suite. However, they did not take care of the priorities that can be pertinent to the refactoring operations in hand, and the refactoring efforts were also disregarded.

Lee et al. [135] applied ordering messy GA (OmeGA) to schedule refactoring of code clones. In formulating the refactoring scheduler, they took into account the probable sequential dependencies among the refactoring activities. However, they did not consider priorities, required efforts, and other types of constraints that may affect the scheduling process. The objective of their approach was to compute a schedule that maximized code quality while maintaining the ordering dependency. For estimating the code quality before and after refactoring, they also used the QMOOD [18] design quality metric suite. In the evaluation of their approach they applied `CCFinder` clone detector and further processed the result to extract only the method/function level clones. However, since `CCFinder` cannot detect *Type-3* clones, their work was mainly based on *Type-1* and *Type-2* clones only.

Zibran and Roy [202, 203] formulated scheduling of code clone refactoring as a constraint satisfaction optimization problem and applied constraint programming (CP) technique to compute an optimal solution of the problem. They discovered and took into account a wide range of hard and soft constraints (in terms of dependencies

and priorities) that can raise in a pragmatic situation for code clone refactoring scheduling. For the assessment of expected gain from the refactoring, they also used the QMOOD [18] design quality metric suite. Moreover, for the estimation of the effort needed to perform different refactoring operations, they proposed the first effort model for code clone refactoring in object-oriented code. Their effort model is an extensive one, which tries to capture the context and various aspects that can affect refactoring efforts. In the evaluation of their effort model and scheduling approach, they carried out case studies using `NiCad` as a code clone detector. Thus, their work took into account not only the *Type-1* and *Type-2* clones, they also dealt with *Type-3* clones.

Table 11: Comparison of Code Clone Refactoring Schedulers

	Bouktif et al. [30]	Lee et al. [135]	Liu et al. [141]	Zibran and Roy [203]
Approach	GA	OmeGa	Heuristic	CP
Refactoring effort	√			√
Quality gain	√	√	√	√
Sequential dependency		√	√	√
Mutual exclusion		√	√	√
Mutual inclusion				√
Priorities satisfaction				√

Table 11 summarizes the different techniques and types of constraints taken care of in the approaches described above. With regards to the scheduling techniques, the evolutionary algorithms such as GA as well as the artificial intelligence (AI) techniques such as heuristic based approaches may suffer from local optima, and do not guarantee optimality. O'Keeffe et al. [158] conducted an empirical comparison of *simulated annealing* (SA), *GA* and *multiple ascent hill-climbing* techniques in scheduling refactoring activities in five software systems written in Java. They reported that among those AI techniques, the hill-climbing approach performed the best. CP is a relatively recent technique that combines the strengths of both AI and OR techniques [20], and thus can be expected to perform better. Nonetheless, an empirical comparison of CP with AI and evolutionary algorithms in optimizing the scheduling of code clone refactoring can be an interesting study.

4.11 Verification of Clone Modification/Refactoring

Human effort in source code modification can be error prone. For example, while renaming identifiers in a copied code, the programmer may mistakenly leave a name unchanged. The work of Jablonski and Hou can be considered a contribution in the verification of clone modification. Their tool `CReN` [86] can check for any inconsistencies in the renaming of identifiers within a code fragment and suggest modifications for making the renaming consistent. `JSync` [154] also offers a similar facility. While `CRen` supports this validation right in the editor inside Eclipse IDE, `JSync` performs the check at the server side, when the code is checked-in to the central repository. Any identified inconsistency in the renaming of identifiers are reported back to the developer with suggestions for attaining consistency.

By definition, refactoring should only alter the structure of the program without changing its behaviour. Therefore, the automatic or semi-automatic refactoring of code clones should be followed by verification to ensure that the refactoring did not change the program behaviour [204]. Automated test case generation and automated adaptation of test cases to the refactored code (clones) may help in this regard. However, to the best of our knowledge, no significant attempt is made towards this possibility.

5 Industrial Adoption of Clone Management

Despite the active research on software clones and their impact on the development and maintenance of software system, management of code clones is still far from wide industrial adoption. A reason to this could be the unavailability of integrated tool support for versatile clone management. Typically, the organizations in the software industry operate on a limited budget and often in tight schedule, when the major objective becomes to be able to deliver the product to the client in time. Cloning becomes an advantage in such scenarios; the immediate effect of code cloning is rewarding, since cloning offers a reuse mechanism for low risk faster development process.

The possible negative impacts of code clones are generally deferred at later stage in the maintenance phase, for instance, until a fault in the system is discovered, which might have been caused due to inconsistent changes in the cloned code, or when it becomes difficult to manage the code base due its very large size. Many software organizations have separate business agreement with the client to provide maintenance support for their product over a defined period of time, for which the client makes additional payments.

Thus, from the business perspective, a software organization may have two phases of business: business on product delivery and product maintenance. Indeed, code

cloning is beneficial at the first phase, though at the second phase there remains a possibility of increase in necessary maintenance effort due to code clones. Since, the software company may consider the maintenance phase a separate cash-inflow business, they might become apathetic in clone management, specially during the active development process. Therefore, the adoption of clone management in the industry largely depends on their realization in the fact that some initial effort in proactive clone management may significantly save later maintenance efforts. There is a saying, "*a stitch in time saves nine*".

6 Conclusion

Software clone research has gained quite some maturity over the last decade, though the majority of the work focused on the detection and analysis of code clones. Compared to those, clone management has earned recent interest due to its pragmatic importance. Notably three surveys [125, 159, 163], none of which focused on clone management, and thus a survey on clone management was a timely necessity. This paper presents a comprehensive survey on clone management and pin-points research achievements and scopes for further work towards a versatile clone management system.

At the fundamental level, the vagueness in the definition of clones at times causes difficulties in formalization, generalization, creation of benchmark data-set, as well as comparison of techniques and tools. A set of task oriented definitions or taxonomies can address these issues. Most of the integrated tools have limitations in detecting *Type-3* clones. Moreover, most of the research on software clones so far emphasized clone analysis at different level of granularity. A variety of techniques for the visualization of clones and the evolution have been proposed. Surprisingly, while the clone analysis points to the importance of considering inheritance hierarchy for extracting clone reengineering candidates, there is still not enough visualization support to analysis clones with respect to their existence in the inheritance hierarchy.

Research on clone management beyond detection has mostly been limited to devising techniques to identify clones that are easier to deal with. In ideal case, simple things should be made easy, while difficult things possible. The state of the art demands more research in semi-automated tool support for clone refactoring and the cost-benefit analysis of clone removal/refactoring. For the integrated tool support for clone management, `JSync` [154] offers a relatively wider set of features compared to others. But, we see a lot more to be done towards a versatile clone management system. Perhaps, due to the unavailability of such tools, there is not much developer-centric ethnographic studies on the patterns of clone management in practice, as well as on the usability and effectiveness of tool support. This survey exposes such potential avenues for further research to create a better impact in the community.

References

[1] E. Adar. GUESS: a language and interface for graph exploration. In *CHI*, pages 791–800. ACM, 2006.

[2] E. Adar and M. Kim. SoftGUESS: Visualization and exploration of code clones in context. In *ICSE*, pages 762 –766, 2007.

[3] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *ICSM*, page 273, 2001.

[4] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755 – 765, 2002.

[5] M. Asaduzzaman. Visualization and analysis of software clones. M.Sc. thesis, University of Saskatchewan, Canada, 2011.

[6] M. Asaduzzaman, C. Roy, and K. Schneider. VisCad: flexible code clone analysis support for NiCad. In *IWSC*, pages 77–78. ACM, 2011.

[7] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *CSMR*, pages 81 –90, 2007.

[8] M. Bahtiyar. JClone : Syntax tree based clone detection for java. Master's thesis, Linnaeus University, 2010.

[9] J. Bailey and E. Burd. Evaluating clone detection tools for use during preventative maintenance. In *SCAM*, pages 36 – 43, 2002.

[10] B. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[11] B. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86 –95, 1995.

[12] B. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28 – 42, 1996.

[13] B. Baker and U. Manber. Deducing similarities in java sources from bytecodes. In *USENIX ATEC*, pages 15–15, Berkeley, CA, USA, 1998. USENIX Association.

[14] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *ICSM*, pages 24–33, 2007.

[15] M. Balazinska, E. Merlo, M. Dagenais, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE*, pages 98–107. IEEE Computer Society Press, 2000.

[16] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS*, pages 292 –303, 1999.

[17] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE*, pages 326–336. IEEE Computer Society, 1999.

[18] J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. on Softw. Engg.*, 28(1):4–17, 2002.

[19] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *ICSM*, pages 273 –282, 2011.

[20] R. Barták. Constraint programming: In pursuit of the holy grail. In *WDS (invited lecture)*, pages 1–10, 1999.

[21] H. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. *SIGSOFT Softw. Eng. Notes*, 30:156–165, 2005.

[22] H. Basit and S. Jarzabek. *Towards Structural Clones: Analysis and semi-automated detection of design-level similarities in software*. VDM Verlag Dr. Müller, 2010.

[23] H. Basit, S. Puglisi, W. Smyth, A. Turpin, and S. Jarzabek. Efficient token based clone

detection with flexible tokenization. In *ESEC-FSE companion*, pages 513–516. ACM, 2007.
[24] H. Basit, D. Rajapakse, and S. Jarzabek. An empirical study on limits of clone unification using generics. In *SEKE*, pages 109–114, 2005.
[25] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, page 368, 1998.
[26] M. Beard. Extending bug localization using information retrieval and code clone location techniques. In *WCRE*, pages 425 –428, 2011.
[27] S. Bellon. Vergleich von techniken zur erkennung duplizierten quellcodes. Diploma thesis, Universität Stuttgart, 2002.
[28] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. on Softw. Engg.*, 33(9):577–591, 2007.
[29] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan. An empirical study on inconsistent changes to code clones at release level. *Science of Computer Programming*, page 17, 2010.
[30] S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo. A novel approach to optimize clone refactoring activity. In *GECCO*, pages 1885–1892, 2006.
[31] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. In *SCAM*, pages 77 –86, 2010.
[32] M. Bruntink. Aspect mining using clone class metrics. In *WARE*, 2004.
[33] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31:804–818, 2005.
[34] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5:46–54, 2007.
[35] P. Bulychev and M. Minea. An evaluation of duplicate code detection using anti-unification. In *IWSC*, 2009.
[36] D. Chatterji, J. Carver, B. Massengil, J. Oslin, and N. Kraft. Measuring the efficacy of code clone information in a bug localization task: An empirical study. In *ESEM*, pages 20–29. IEEE Computer Society, 2011.
[37] A. Chiu and D. Hirtle. Beyond clone detection. CS846 Course Project Report, University of Waterloo, 2007.
[38] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano. Extracting code clones for refactoring using combinations of clone metrics. In *IWSC*, pages 7–13. ACM, 2011.
[39] J. Cordy. Comprehending reality: Practical barriers to industrial adoption of software maintenance automation. In *IWPC*, pages 196–206. IEEE Computer Society, 2003.
[40] J. Cordy. Live scatterplots. In *IWSC*, pages 79–80. ACM, 2011.
[41] J. Cordy, T. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *CASCON*, pages 1–12. IBM Press, 2004.
[42] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3/4):219 – 236, 1995.
[43] I. Davis and M. Godfrey. Clone detection by exploiting assembler. In *IWSC*, pages 77–78. ACM, 2010.
[44] I. Davis and M. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *WCRE*, pages 242 –246, 2010.
[45] M. de Wit. *Managing Clones Using Dynamic Change Tracking and Resolution*. M.Sc. thesis, Delft University of Technology, 2008.
[46] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model clone detection in practice. In *IWSC*, pages 57–64. ACM, 2010.
[47] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE*, pages 603–612. ACM,

2008.
[48] G. Di Lucca, M. Di Penta, and A. Fasolino. An approach to identify duplicated web pages. In *COMPSAC*, pages 481 – 486, 2002.
[49] E. Duala-Ekoko and M. Robillard. Tracking code clones in evolving software. In *ICSE*, pages 158–167, 2007.
[50] E. Duala-Ekoko and M. Robillard. CloneTracker: tool support for code clone management. In *ICSE*, pages 843–846, 2008.
[51] E. Duala-Ekoko and M. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.*, 20:3:1–3:31, 2010.
[52] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109 –118, 1999.
[53] S. Ducasse, M. Rieger, G. Golomingi, and B. Bym. Tool support for refactoring duplicated OO code. In *ECOOP Workshop Reader*, number 1743 in LNCS, pages 2–6. Springer-Verlag, 1999.
[54] W. Evans, C. Fraser, and F. Ma. Clone detection via structural abstraction. In *WCRE*, pages 150–159. IEEE Computer Society, 2007.
[55] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Softw. Engg.*, 13:601–643, 2008.
[56] R. Fanta and V. Rajlich. Removing clones from the code. *Journal of Software Maintenance: Research and Practice*, 11(4):223–243, 1999.
[57] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, 1987.
[58] M. Fowler, K. Beck, J.Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.
[59] Y. Fukushima, R. Kula, S. Kawaguchi, K. Fushida, M. Nagura, and H. Iida. Code clone graph metrics for detecting diffused code clones. In *APSEC*, pages 373 –380, 2009.
[60] M. Funaro, D. Braga, A. Campi, and C. Ghezzi. Combining syntactic and textual approach in clone detection. In *IWSC*, 2010.
[61] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330. ACM, 2008.
[62] S. Giesecke. Generic modelling of code clones. In *DRSS*, pages 1–23, 2007.
[63] S. Giesecke. Dupman - eclipse duplication management framework, http://sourceforge.net/projects/dupman/, last access: Dec 2011.
[64] N. Göde. Incremental clone detection. Diploma thesis, University of Bremen, 2008.
[65] N. Göde. Evolution of type-1 clones. In *SCAM*, pages 77–86, 2009.
[66] N. Göde and J. Harder. Clone stability. In *CSMR*, pages 65 –74, 2011.
[67] N. Göde and R. Koschke. Incremental clone detection. In *CSMR*, pages 219–228, 2009.
[68] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 1–28, 2010.
[69] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *ICSE*, pages 311–320. ACM, 2011.
[70] J. Harder and N. Göde. Modeling clone evolution. In *IWSC*, pages 17–21, 2009.
[71] J. Harder and N. Göde. Quo vadis, clone management? In *IWSC*, pages 85–86. ACM, 2010.
[72] J. Harder and N. Göde. Efficiently handling clone data: Rcf and cyclone. In *IWSC*, pages 81–82. ACM, 2011.
[73] J. Helfman. Dotplot patterns: a literal look at pattern languages. *Theor. Pract. Object Syst.*, 2:31–41, 1996.
[74] A. Hemel, K. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *MSR*, pages 63–72. ACM, 2011.

[75] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In *IASTED-SEA*, pages 222–229. ACTA Press, 2004.
[76] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. *PROFES*, (LNCS 3009):220–233, 2004.
[77] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Inf. Softw. Technol.*, 49:985–998, 2007.
[78] Y. Higo and S. Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *WCRE*, pages 315 –316, 2009.
[79] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On software maintenance process improvement based on code clone analysis. In *PROFES*, pages 185–197. Springer-Verlag, 2002.
[80] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous modification support based on code clone analysis. In *APSEC*, pages 262–269. IEEE Computer Society, 2007.
[81] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: A PDG-based approach. In *WCRE*, pages 3 –12, 2011.
[82] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software. In *IWPSE-EVOL*, pages 73–82. ACM, 2010.
[83] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *ICPC*, pages 238–242, 2009.
[84] D. Hou, F. Jacob, and P. Jablonski. Exploring the design space of proactive tool support for copy-and-paste programming. In *CASCON*, pages 188–202. ACM, 2009.
[85] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *ICSM*, pages 1 –9, 2010.
[86] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *ETX*, pages 16–20, 2007.
[87] P. Jablonski and D. Hou. Renaming parts of identifiers consistently within code clones. In *ICPC*, pages 38–39. IEEE Computer Society, 2010.
[88] F. Jacob, D. Hou, and P. Jablonski. Actively comparing clones inside the code editor. In *IWSC*, pages 9–16. ACM, 2010.
[89] K. Jalbert and J. Bradbury. Using clone detection to identify bugs in concurrent software. In *ICSM*, pages 1 –5, 2010.
[90] S. Jarzabek and S. Li. Unifying clones with a generative programming technique: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(4):267–292, 2006.
[91] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.
[92] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE*, pages 55–64. ACM, 2007.
[93] Z. Jiang and A. Hassan. A framework for studying clones in large software systems. In *SCAM*, pages 203 –212, 2007.
[94] Z. Jiang, A. Hassan, and R. Holt. Visualizing clone cohesion and coupling. In *APSEC*, pages 467 –476, 2006.
[95] J. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*, pages 171–183. IBM Press, 1993.
[96] J. Johnson. Substring matching for clone detection and change tracking. In *ICSM*, pages 120 –126, 1994.
[97] J. Johnson. Visualizing textual redundancy in legacy source. In *CASCON*, pages 32–41. IBM Press, 1994.
[98] J. Johnson. Navigating the textual redundancy web in legacy source. In *CASCON*, pages 16–25. IBM Press, 1996.

[99] E. Juergens. Research in cloning beyond code: a first roadmap. In *IWSC*, pages 67–68. ACM, 2011.
[100] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can clone detection support quality assessments of requirements specifications? In *ICSE*, volume 2, pages 79 –88, 2010.
[101] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can clone detection support quality assessments of requirements specifications? In *ICSE*, pages 79 –88, 2010.
[102] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective - a workbench for clone detection research. In *ICSE*, pages 603–606, 2009.
[103] E. Juergens, B. Hummel, F. Deissenboeck, and M. Feilkas. Static bug detection through analysis of inconsistent clones. In *TESO*, pages 443–446, 2008.
[104] N. Juillerat and B. Hirsbrunner. An algorithm for detecting and removing clones in java code. In *SeTra*, pages 63–74, 2006.
[105] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.
[106] C. Kapser. *Toward an Understanding of Software Code Cloning as a Development Practice*. PhD thesis, University of Waterloo, Canada, 2009.
[107] C. Kapser and M. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE*, pages 85–94. IEEE Computer Society, 2004.
[108] C. Kapser and M. Godfrey. Improved tool support for the investigation of duplication in software. In *ICSM*, pages 305–314. IEEE Computer Society, 2005.
[109] C. Kapser and M. Godfrey. "Cloning considered harmful" considered harmful. In *WCRE*, pages 19–28, 2006.
[110] C. Kapser and M. Godfrey. Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13:645–692, 2008.
[111] Cory J. Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems: A case study. *J. Softw. Maint. Evol.*, 18:61–82, 2006.
[112] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: A tool for automatic code clone detection in the IDE. In *WCRE*, pages 313–314, 2009.
[113] A. Kellens, K. Mens, and P. Tonella. In A. Rashid and M. Aksit, editors, *Transactions on aspect-oriented software development IV*, chapter A survey of automated code-level aspect mining techniques, pages 143–162. Springer-Verlag, 2007.
[114] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.
[115] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.
[116] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *ICSE*, pages 301–310. ACM, 2011.
[117] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR*, pages 1–5. ACM, 2005.
[118] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.
[119] E. Kodhai, V. Vijayakumar, G. Balabaskaran, T. Stalin, and B. Kanagaraj. Method level detection and removal of code clones in C and Java programs using refactoring. In *IJJCET*, pages 93–95. Gopalax Publications & TCET, 2010.
[120] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56. Springer-Verlag, 2001.

[121] G. Koni-N'Sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Bern, 2001.
[122] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *WCRE*, pages 44 –54, 1997.
[123] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for design concept localization. In *WCRE*, pages 96 –103, 1995.
[124] K. Kontogiannis, R. Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Autom. Softw. Eng.*, 3(1/2):77–108, 1996.
[125] R. Koschke. Survey of research on software clones. In *DRSS*, pages 1–24, 2006.
[126] R. Koschke. Frontiers of software clone management. In *FoSM*, pages 119 –128, 2008.
[127] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE*, pages 253–262, 2006.
[128] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301 –309, 2001.
[129] J. Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE*, pages 170 –178, 2007.
[130] J. Krinke. Is cloned code more stable than non-cloned code? *SCAM*, 0:57–66, 2008.
[131] Jens Krinke. Is cloned code older than non-cloned code? In *IWSC*, pages 28–33. ACM, 2011.
[132] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, pages 314–321. IEEE Computer Society, 1997.
[133] F. Lanubile and T. Mallardo. Finding function clones in web applications. In *CSMR*, pages 379–286. IEEE Computer Society, 2003.
[134] M. Lee, J. Roh, S. Hwang, and S. Kim. Instant code clone search. In *FSE*, pages 167–176, 2010.
[135] S. Lee, G. Bae, H. Chae, D. Bae, and Y. Kwon. Automated scheduling for clone-based refactoring using a competent ga. *Softw. Pract. Exper.*, 41(5):521–550, 2010.
[136] S. Lee and I. Jeong. SDD: high performance code clone detection system for large scale source code. In *OOPSLA*, pages 140–141, 2005.
[137] A. Leitão. Detection of redundant code using r2d2. *Software Quality Control*, 12:361–382, 2004.
[138] H. Li and S. Thompson. Similar code detection and elimination for Erlang programs. *Practical Aspects of Declarative Languages*, 5937:104–118, 2010.
[139] H. Li and S. Thompson. Incremental clone detection and elimination for erlang programs. In *FASE/ETAPS*, pages 356–370. Springer-Verlag, 2011.
[140] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176 – 192, 2006.
[141] H. Liu, G. Li, Z. Ma, , and W. Shao. Conflict-aware schedule of software refactorings. *IET Software*, 2(5):446–460, 2008.
[142] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *APSEC*, pages 269 –276, 2006.
[143] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Analysis of the linux kernel evolution using code clone coverage. In *MSR*, page 22, 2007.
[144] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *ICSE*, pages 106–115, 2007.
[145] A. Lozano and M. Wermelinger. Tracking clones' imprint. In *IWSC*, 2010.
[146] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *ASE*, pages 107 – 114, 2001.

[147] J. Mayrand, B. Lague, and J. Hudepohl. Evaluating the benefits of clone detection in the software maintenance activities in large scale systems. In *WESS*, 1996.
[148] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244 –253, 1996.
[149] R. Miller. *Lightweight Structured Text Processing*. PhD thesis, Carnegie Mellon University, 2001.
[150] R. Miller and B. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX*, pages 161–174. USENIX Association, 2001.
[151] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *ACM-SAC (SE Track)*, pages 1–8, 2012 (to appear).
[152] E. Murphy-Hill, P. Quitslund, and A. Black. Removing duplication from java.io: a case study using traits. In *OOPSLA*, pages 282–291. ACM, 2005.
[153] S. Nasehi, G. Sotudeh, and M. Gomrokchi. Source code enhancement using reduction of duplicated code. In *IASTED*, pages 192–197. ACTA Press, 2007.
[154] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone management for evolving software. *IEEE Trans. on Softw. Engg.*, 1(1):1–19, 2011.
[155] T. Nguyen, H. Nguyen, J. Al-Kofahi, N. Pham, and T. Nguyen. Scalable and incremental clone detection for evolving software. In *ICSM*, pages 491 –494, 2009.
[156] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Cleman: Comprehensive clone group evolution management. In *ASE*, pages 451–454, 2008.
[157] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone-aware configuration management. In *ASE*, pages 123–134, 2009.
[158] M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.: Res. Pract.*, 20:345–364, 2008.
[159] J. Pate, R. Tairas, and N. Kraft. Clone evolution: a systematic review. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 1–23, 2011.
[160] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, and T. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE*, pages 276–286. IEEE Computer Society, 2009.
[161] M. Rieger. *Effective Clone Detection Without Language Barriers*. Phd thesis, Institut fuï Informatik und angewandte Mathematik, Germany, 2005.
[162] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109, 2004.
[163] C. Roy and J. Cordy. A survey on software clone detection research. Tech Report TR 2007-541, School of Computing, Queens University, Canada, 2007.
[164] C. Roy and J. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.
[165] C. Roy and J. Cordy. Scenario-based comparison of clone detection techniques. In *ICPC*, pages 153–162, 2008.
[166] C. Roy and J. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *ICSTW*, pages 157–166, 2009.
[167] C. Roy and J. Cordy. Near-miss function clones in open source software: an empirical study. *J. of Softw. Maintenance and Evolution: Research and Practice*, 22(3):165–189, 2010.
[168] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74:470–495, 2009.
[169] V. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *ASE*, pages 336 – 339, 2004.
[170] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *ISSTA*, pages 117–128. ACM, 2009.

[171] R. Saha. Detection and analysis of near-miss clone genealogies. M.Sc. thesis, University of Saskatchewan, 2011.
[172] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *SCAM*, pages 87–96, 2010.
[173] R. Saha, C. Roy, and K. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *ICSM*, pages 293 –302, 2011.
[174] R. Saha, C. Roy, and K. Schneider. Visualizing the evolution of code clones. In *IWSC*, pages 71–72. ACM, 2011.
[175] A. Santone. Clone detection through process algebras and java bytecode. In *IWSC*, pages 73–74. ACM, 2011.
[176] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *ECOOP*, volume 2743 of *LNCS*, pages 327–339. Springer Berlin / Heidelberg, 2003.
[177] S. Schulze and M. Kuhlemann. Advanced analysis for code clone removal. In *WSR*, pages 1–2, 2009.
[178] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a refactoring guideline using code clone classification. In *WRT*, pages 6:1–6:4. ACM, 2008.
[179] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, and Y. Zou. Studying the impact of clones on software defects. In *WCRE*, pages 13 –21, 2010.
[180] H. Störrle. Towards clone detection in UML domain models. In *ECSA*, pages 285–293. ACM, 2010.
[181] R. Tairas. Code clones literature, http://students.cis.uab.edu/tairasr/clones/literature/, last access: Feb 2012.
[182] R. Tairas and J. Gray. Phoenix-based clone detection using suffix trees. In *ACM-SE*, pages 679–684, 2006.
[183] R. Tairas and J. Gray. Get to know your clones with CeDAR. In *OOPSLA*, pages 817–818, 2009.
[184] R. Tairas and J. Gray. Sub-clone refactoring in open source software artifacts. In *SAC*, pages 2373–2374. ACM, 2010.
[185] R. Tairas, J. Gray, and I. Baxter. Visualization of clone detection results. In *OOPSLA-ETX*, pages 50–54. ACM, 2006.
[186] R. Tairas, J. Gray, and I. Baxter. Visualizing clone detection results. In *ASE*, pages 549–550. ACM, 2007.
[187] S Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15:1–34, 2010.
[188] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *VLHCC*, pages 173–180. IEEE Computer Society, 2004.
[189] R. Torres. Source code mining for code duplication refactorings with formal concept analysis. M.Sc. thesis, Vrije Universiteit Brussel, Belgium, 2004.
[190] M. Uddin, C. Roy, K. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *WCRE*, pages 13 –22, 2011.
[191] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Code clone analysis tool. In *ISESE*, volume 2, pages 31–32, 2002.
[192] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *METRICS*, pages 67–76. IEEE Computer Society Press, 2002.
[193] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *APSEC*, pages 327 – 336, 2002.
[194] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, pages 128 –135, 2004.

[195] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *WCRE*, pages 285–294. IEEE Computer Society, 2003.

[196] A. Walenstein and A. Lakhotia. The software similarity problem in malware analysis. In *DRSS*, pages 1–10, 2006.

[197] V. Weckerle. CPC: an eclipse framework for automated clone life cycle tracking and update anomaly detection. Master's thesis, Freie Universität Berlin, Germany, 2008.

[198] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21:739–755, 1991.

[199] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On refactoring support based on code clone dependency relation. In *METRICS*, pages 16–25. IEEE Computer Society, 2005.

[200] Y. Zhang, H. Basit, S. Jarzabek, D Anh, and M. Low. Query-based filtering and graphical view generation for clone analysis. In *ICSM*, pages 376 –385, 2008.

[201] M. Zibran. Analysis and management of code clones. In *ICSM (doctoral symposium)*, pages 1–4, 2011.

[202] M. Zibran and C. Roy. Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach. In *ICPC*, pages 266 – 269, 2011.

[203] M. Zibran and C. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *SCAM*, pages 105–114, 2011.

[204] M. Zibran and C. Roy. Towards flexible code clone detection, management, and refactoring in IDE. In *IWSC*, pages 75–76, 2011.

[205] M. Zibran and C. Roy. IDE-based real-time focused search for near-miss clones. In *ACM-SAC (SE Track)*, pages 1–8, 2012 (to appear).

[206] M. Zibran, R. Saha, M. Asaduzzaman, and C. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *ICECCS*, pages 295–304, 2011.