Abstract

Ad-hoc reuse through copy-and-paste occurs frequently in practice affecting the evolvability of software. This paper summarizes the state of the art in software clone management (detection, tracking, presentation, assessing, removal, changing) and the empirical knowledge we have gained so far. In the course of the summary, the paper identifies further research potential.

1. Introduction

Frederick P. Brooks wrote in his famous essay on software engineering [30]:

“Software entities are more complex for their size than perhaps any other human construct because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into a subroutine — open or closed. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.”

Reality is different though. Similar pieces of software – often called clones – occur rather frequently. Several authors report on 7-23% code duplication [7, 56, 62]; in one extreme case even 59% [25]. Facing reality with warts and all, we need to accept clones as a fact of live and learn how to deal with them.

Software clone management comprises all activities of looking after and making decisions about consequences of copying and pasting. This type of redundancy occurs in many artefacts such as data, data base schemes, requirement specifications, architectures, test cases, and code.

Software clone management aims at identifying and organizing existing clones, controlling growth and dispersal of clones, and avoiding clones altogether. Lague et al. [62] and Giesecke [32] distinguish three main lines of clone management:

- preventive clone management (also known as preventive control [62]) comprises activities to avoid new clones
- compensative clone management (also known as problem mining [62]) encompasses activities aimed at limiting the negative impact of existing clones that are to be left in the system
- corrective clone management covers activities to remove clones from a system

To make informed decisions in clone management, we need to understand the reasons for clones, their risks and benefits, and ways to detect, track, present, remove, or consistently change them.

This paper summarizes the state of the art in clone management and gives further directions for research paths.

While copying and pasting occurs in all software artefacts, we will focus on clones in programs in this paper.

The material presented here is largely based on the outcome of the Dagstuhl seminar No. 06301 on Duplication, Redundancy, and Similarity in Software and on my own research with colleagues. Additional more detailed surveys that do not have the ten-page limit were published by myself [57, 58] and Roy and Cordy [79].

2 The Concept of Clones

Unlike in data base theory (the various normal forms), there is no formal concept of redundancy in programs. Our field is still searching for a suitable definition of software clones. Ira Baxter’s definition of clones expresses this vagueness (2002):

“Clones are segments of code that are similar according to some definition of similarity.”

This definition is intentionally vague by not stating a definition of similarity. A brief probe of the world’s expert in software clone management at the Dagstuhl seminar 06301 “Duplication, Redundancy, and Similarity in
Software” 2007 revealed the discrepancies about the notions of a clone [46]. Less than half of the clone candidates Cory Kapser presented to these experts had 80% agreement amongst the judges. Judges appeared to differ primarily in their criteria for judgment rather than their interpretation of the clone candidates. A study by Walenstein et al. had indicated this disagreement before [94]. In this study clones were to be identified that ought to be removed. The human raters of the clones proposed by automated tools did rarely agree. The reason for the disagreement may partly be attributed to the specific guidelines to find clones worth to be removed. Yet, the question remains whether there is a task-independent definition.

Similarity can be based on program text (in its form of pure text, lexical or syntactic structure) or on semantics. Program text similarity may be defined in terms of text, tokens, syntax, or data and control dependencies. Current research focuses on similar program text because detecting semantic similarity is undecidable in general – although semantic similarity is a frequent problem, too. Programmers often reinvent the wheel instead of just using existing code because they are not aware of this code. Also, in cases of plagiarism, code is modified to blur the act of copying and pasting.

Program-text clones can be compared on the basis of the program text that has been copied. We will first consider pairs of code fragments as clones. We can distinguish the types of clone pairs primarily by the transformation that maps one fragment onto the other fragment to make them equal (in brackets, we state the type according to an earlier categorization frequently used [15]). We can then define two additional properties. One relates to the transformation: whether the transformation is a one-to-one mapping or not. The other property relates to the lexical contiguity. These two properties are orthogonal.

**Exact clone (type 1)** is an exact copy of consecutive code fragments without modifications (except for whitespace and comments when insignificant from the perspective of the language definition). That is, the transformation is the identity. For instance, the two fragments in Figure 1a and 1b form a type-1 clone.

**Parameter-substituted clone (type 2)** is a copy where only parameters (identifiers or literals) have been changed; given a suitable parameter substitution, the transformed copy is a type-1 clone.

**Structure-substituted clone** is a copy where program structures (complete subtrees in the syntax tree) have been changed. Given a suitable structure substitution, the transformed copy is a type-1 clone [90].

For parameter-substituted clones, we can replace one leaf in the syntax tree by another leaf. For structure-substituted clones, larger subtrees can be substituted.

For instance, the two fragments in Figure 1a and 1e form a structure-substituted clone by transforming 1 into \texttt{foo}(4).

Note that these three classes form an inclusion hierarchy. Every exact clone is a parameter-substituted clone; every parameter-substituted clone is a structure-substituted clone. There are two variants of parameter and structure substitutions.

**Consistently substituted clone** is a pair of code fragments that can be transformed into each other by substituting via a one-to-one function whose domain is the set of substitution parameters occurring in one fragment and whose range is the set of substitution parameters occurring in the other fragment [8]. For instance, the two fragments in Figure 1a and 1c form a consistently parameter-substituted clone. Similarly, the two fragments in Figure 1a and 1e form a consistently structure-substituted clone.

**Inconsistently substituted clone** is a copy in which the parameter substitution is inconsistent. For instance, the two fragments in Figure 1a and 1d form an inconsistently parameter-substituted clone because the second parameter \texttt{y} in line 12 should have been \texttt{x} to be consistent. Similarly, the two fragments in Figure 1a and 1f form an inconsistently structure-substituted clone because the 1 in line 1 is replaced by \texttt{foo}(4) in line 13 but the 1 in line 2 is not replaced by \texttt{foo}(4) but by 1 in line 14.

We can distinguish the above clone types by another property, namely, lexical contiguity.

**Contiguous clone** is a copy whose parts are all lexically consecutive, that is, there are no other code fragments within the clone fragments that cannot be structurally substituted to form a structure-substituted clone. All clone examples presented so far were contiguous clones.

**Non-contiguous clone (type 3)** is a copy whose parts are not lexically consecutive. By deleting code fragments in both fragments, we can transform them into a contiguous clone.

The above categorization differs from an earlier categorization [15]. Type-3 clones were defined in the earlier categorization as a copy with further modifications where statements were changed, added, or removed. This type of clone corresponds roughly to non-contiguous clones in our classification but differs in the scope of the clone. While
the earlier categorization classifies the whole fragment as a clone, we consider only the equal and similar parts between the two fragments as clone. For instance, while we consider the fragment consisting of lines 16, 17, and 19 as a cloned fragment to lines 1–3, the earlier categorization would consider lines 16–19 including the difference as the clone. The advantage of our classification is that the clone relation is transitive, while type-3 clones as defined earlier may not be transitive.

Given the transitivity of the clone relations, clone pairs may be summarized to clone equivalence classes. However, we should be aware that transitivity does not hold when clones should be maximally long [6]. A clone pair is maximally long if its two fragments cannot be extended to the left nor to the right so that the extensions form a clone pair, too. For instance, if we have the following string modelling code fragments adbdadb, we observe that d occurs three times in this string. The second and third d form a maximally long clone pair, but the first and last d do not form a maximally long clone pair because they can be extended to the left and right to form the subsuming clone pair of the two abc.

Our classification is based on transformation, consistency, and contiguity but not on content. Balazinska et al. introduced a more refined classification for function clones [10] by distinguishing the type of differences between structure-substituted clones. This classification makes sense for selecting a suitable strategy for clone removal.

Another classification by Kapser et al.'s distinguishes by scope (within or across a file), type of region at file level (functions, global declarations, macros), degree of overlap or containment, and type of code sequence [50, 48, 47].

The many types of similarities of programs that may be used to distinguish types of clones were summarized by Walenstein et al. [93].

Clearly, the definition of redundancy, similarity, and cloning in software is still an open issue. Walenstein has summarized the debate on terminology among the clone experts at the mentioned Dagstuhl seminar [92]. There is little consensus in this matter. One problem is that the viewpoint of what constitutes a clone may depend upon a particular task and other constraints. For instance, in a refactoring exercise, we want to find only syntactic clones that may be replaced by the available means of the programming language. On the other hand, we are interested in every type of copy when we modified a piece of code and need to check whether the same change must be replicated somewhere else.

### 3 The "Nature" of Code Clones

For the informed management of clones, we need to know how they come into existence, how they evolve, and their pros and cons.

#### 3.1 How Clones Come About

Kim et al. [51] observed programmers in their daily practice to find out why programmers copy and paste code. They attribute the reasons for cloning to (1) limitations of the programming language, (2) deferred generalization: code is copied and modified and only then the differences are factored out when they are all known and understood, (3) cross-cutting concerns, and (4) templating: copied text is used as a template and then customized in the pasted context. This study is still a very first field observation with specific constraints. For instance, the observed programmers were researchers at IBM T.J. Watson and it is not clear how representative they are.

Another field study was conducted by Kapser and Godfrey investigating clones in large systems [49]. They found what they call patterns of cloning where cloning is consciously used as an implementation strategy. In their case study, they found the following cloning patterns:

**Forking** is cloning used to bootstrap development of similar solutions, with the expectation that evolution of the
code will occur somewhat independently, at least in the short term.

**Templating** is used as a method to directly copy behavior of existing code but appropriate abstraction mechanisms are unavailable. This kind of pattern was detected by Kim et al. [51], too.

**Customization** occurs when currently existing code does not adequately meet a new set of requirements. The existing code is cloned and tailored to solve this new problem.

Very likely other more organizational aspects play a role, too, such as time pressure, unavailable information on the impact of code changes, or even inadequate performance measures of programmers' productivity. We need more empirical studies of this type to understand the reasons for cloning better.

### 3.2 How they Evolve

There are a several empirical studies on the evolution of clones, which describe some interesting observations. Early studies by Antoniol et al. indicate that it is possible to predict the evolution of clones based on the past [2] and that the scope of cloning is typically limited to subsystems [3]. An independent study by Godfrey and Tu confirms the last observation and found that cloning is common and steady practice in the Linux kernel [35]. They found that the cloning rate does increase steadily over time but not in peaks. The study by Li et al. [66] observed for the Linux kernel in the period of 1994 to 2004 that the redundancy rate has increased from about 17% to about 22%. Interestingly, most of the growth of redundancy rate comes from a few subsystems. That is, different subsystems may be differently clone-sensitive. Another study by Kim et al. showed that many code clones exist in the system for only a short time [52]. Kim et al. conclude that extensive refactoring of such short-lived clones may not be worthwhile if they likely diverge from one another very soon. Moreover, many long-lasting clones that are not changed consistently cannot easily be avoided because of limitations of the programming language.

Concerning the consistency of changes on clone code there are currently two dissenting studies. A study by Aversano et al. suggests for both bug fixing and for evolution purposes, most of the cloned code is consistently maintained during the same cochange or during cochanges close in time [4]. A later study by Krinke showed that clone groups are consistently changed in roughly only half of the time [61]. The study also showed that later changes do not make up for missed changes in general, because inconsistently changed clone groups that become consistently changed clone groups later can be found rarely. Further studies are required to investigate these conflicting results.

Al-Ekram et al. have looked at clones between different systems of the same domain (text editors and window managers) to see whether programmers copy from other systems [1]. They did not find evidence for cross-system cloning but found similarities that stems from the underlying protocols of the libraries these systems share.

### 3.3 Their Pros and Cons

Several researchers have looked into the pros and cons of clones. The current debate is based mostly on plausible arguments, as for instance, that changes must be made consistently multiple times if code is redundant, while on the other hand, organizational issues may favor cloning [21], for instance, to disentangle development units. Likewise, the cloning patterns by Kapser and Godfrey use code cloning as a purposeful implementation strategy which may make sense under certain circumstances [49].

There are initial empirical studies that explore the interrelationship of code cloning and maintainability. These studies may help to base the plausible arguments on empirical evidence. All of them focus on code cloning and errors and change frequency as one (out of many) maintainability aspects, however.

For a large and old COBOL program, Monden et al. [74] found correlations of maximal clone length with change frequency and number of errors. They found that most errors were reported for modules with clones of at least 200 lines. They also found many errors – although less than in those with longer clones – in modules with shorter clones up to 50 lines. Yet, interestingly enough, they found the lowest error rate for modules with clones of 50 to 100 lines.

Chou et al. [18] investigated the hypothesis that if a function, file, or directory has one error, it is more likely that it has others. They found in their analysis of the Linux and OpenBSD kernels that this phenomenon can be observed most often where programmer ignorance of interface or system rules combines with copy-and-paste.

Li et al. [66] use clone detection to find bugs when programmers copy code but rename identifiers in the pasted code inconsistently. On average, 13% of the clones flagged as copy-and-paste bugs by their technique turned out to be real errors. In 73% the flagged potential bugs were not bugs on average. The remaining potential problems are still under analysis by the developers of the analyzed systems. Similar results were reported by independent other studies [44, 9].

A preliminary study by Lozano et al. suggests that functions with cloned code change more and more frequently [67]. In a larger study by Geiger et al. many cases were found in which clones had to be updated frequently, al-
**Textual**

- [41, 42] hashing of strings per line, then textual comparison
- [25] hashing of strings per line, then visual comparison using dotplots
- [70] latent semantic indexing for identifiers and comments
- [80] syntactic pretty-printing and normalization, then textual comparison between lines

**Lexical**

- [7, 8] suffix trees for tokens per line
- [45] token normalizations, then suffix-tree based search
- [65] data mining for frequent token sequences

**Metrics**

- [62, 56, 71, 55] comparing metrics for functions
- [24, 63] comparing metrics for web sites

**Syntax**

- [13, 98] hashing of syntax trees and tree comparison
- [91] data mining for frequent syntax subtrees
- [59, 26] serialization of syntax trees and suffix-tree detection
- [90] derivation of syntax patterns and pattern matching
- [40] metrics for syntax trees and metric vector comparison

**Data and Control Dependency**

- [60, 53] approximative search for similar subgraphs in PDGs

---

Figure 2: Clone Detection Techniques

though the study did not reveal a statistically significant correlation between clones and change frequency.

### 4 Clone Management

This section describes methods and techniques to detect, present, remove, prevent, and compensate clones.

#### 4.1 Detection

Automated clone detection is a necessary part of clone management to locate and track clones. This section summarizes the state of the art in this area. The techniques can be distinguished primarily in the type of information their analysis is based on and the types of techniques they are using (cf. Figure 2).

**Textual comparison:** The simplest way to detect clones is to compare program text textually. The advantage is that the technique can be applied to any kind of programming language. The disadvantage is that minor textual modifications, for instance, changes in identifiers or layout, may mislead the analysis.

In order to overcome sensitivity to layout changes of textual approaches, Roy and Cordy suggest to parse, normalize, and pretty-print the code before the textual comparison [80], but then the technique requires a syntax analysis.

**Token comparison:** A lexical analysis turns a program into a stream of tokens (indivisible units of meaning). Clone detection then turns into the problem of finding similar token subsequences. Most often, these techniques use suffix trees [72] for the search. Because of the space and time complexity linear to the program length, these approaches scale very well. Also, lexical analysis is relatively simple, so that the technique can quickly be adjusted to other languages. Because only lexical analysis is used, the code does not need to be complete or syntactically correct. The disadvantage is that the technique finds many clones that are not syntactic units. Either in a preprocessing [83, 22, 33] or post-processing [37] step, clones that completely fall in syntactic blocks can be found if block delimiters are known. Preprocessing and postprocessing both require some syntactic information—gathered either lightweight by counting tokens opening and closing syntactic scopes or island grammars [75] or a full-fledged syntax analysis [33].

**Metric comparison:** Instead of comparing program text directly, we can gather metric vectors for the program text and then compare the metric vectors. Close metric vectors (e.g., measured by the Euclidean distance) indicate similar code. Because it is practically infeasible to gather metrics for every statement sequence of arbitrary length, these techniques typically compare only whole functions. A selection of independent, broad, and significant metrics is essential.

**Comparison of syntax trees:** Parsing yields a syntax tree that can be searched for similar subtrees. The techniques of this category find syntactic units as clones. The disadvantage is the necessity to develop a parser and the overhead for parsing.

**Comparison of program dependency graphs (PDG):** Even the syntax-based techniques can be mislead by simple program transformations such as swapping two lines. Syntax-based clone detection has difficulties to find non-contiguous clones. A program dependency graph (PDG) [28] is a representation of a program that represents only the control and data dependency among statements. This way program dependency graphs abstract from the textual order. Clones may then be identified as isomorphic subgraphs in a program dependency graph [60, 53]. Because this problem is NP hard, the algorithms use approximative solutions. The disadvantage of these techniques is that the analysis is quite expensive.
Other techniques: Leitao [64] combines syntactic and semantic techniques through a combination of specialized comparison functions that compare various aspects (similar call subgraphs, commutative operators, user-defined equivalences, transformations into canonical syntactic forms). Each comparison function yields an evidence that is summarized in an evidence-factor model yielding a clone likelihood.

Comparison of Clone Detection Algorithms: Existing clone detectors can be compared in terms of recall and precision of their findings as well as suitability for a particular purpose. There are several evaluations along these lines based on qualitative and quantitative data.

The most comprehensive quantitative study to date is the evaluation by Bellon et al. [15]. Their findings for the tools listed in Figure 3a are summarized in Figure 3b. This study confirmed a similar earlier, but smaller study by Bailey and Burd [5] with the exception of precision and recall of the metric-based technique. The better precision of Merlo’s technique in the study by Bellon et al. can be explained by the fact that Merlo used a token-based technique additionally to detect type-1 and type-2 clones.

<table>
<thead>
<tr>
<th>No.</th>
<th>Participant</th>
<th>Tool</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brenda S. Baker [7]</td>
<td>Dup</td>
<td>Token</td>
</tr>
<tr>
<td>2</td>
<td>Ira D. Baxter [13]</td>
<td>CloneDr</td>
<td>AST</td>
</tr>
<tr>
<td>3</td>
<td>Toshihiro Kamiya [45]</td>
<td>CCFinder</td>
<td>Token</td>
</tr>
<tr>
<td>4</td>
<td>Jens Krinke [60]</td>
<td>Duplix</td>
<td>PDG</td>
</tr>
<tr>
<td>5</td>
<td>Ettore Merlo [71]</td>
<td>CLAN</td>
<td>Metrics, Token</td>
</tr>
<tr>
<td>6</td>
<td>Matthias Rieger [25]</td>
<td>Duploc</td>
<td>Text</td>
</tr>
</tbody>
</table>

(a) Participating scientists

(b) Summary of Results

While the Bailey/Burd and Bellon/Koschke studies focus on quantitative evaluation of clone detectors, other authors have evaluated clone detectors for the fitness for a particular maintenance task. Rysselbergh and Demeyer [89] compared text-based [25, 77], token-based [7], and metric-based [71] clone detectors for refactoring. They compare these techniques in terms of suitability, relevance, confidence, and focus qualitatively.

Roy and Cordy evaluate clone detection tools by their estimated potential to accurately detect various types of clones [81]. The clones selected for this thought experiment cover the categories introduced in Section 2 representing hypothetical copying and editing scenarios.

4.2. Presentation

Because of the large and complex information space of cloning and the necessity for human interpretation, suitable means are required to make this information space accessible to human analysts. Various levels of system descriptions must be combined: code, architecture, and requirements. Visualization is one means to provide such a holistic view. The types of visualization proposed specifically for clone information are summarized in Figure 4. All these types of visualization are standard techniques not invented for clone information specifically. The challenge is to integrate these standard techniques intelligently and to provide empirical evidence through experimentation that they are actually useful.

4.3. Correction

There are many ways to remove clones depending upon the type of clones and the underlying programming language. Both research prototypes and commercial tools, such as CloneDr\textsuperscript{1} by Semantic Designs, automatically detect and remove clones.

In simple cases, you can use functional abstraction to replace equivalent copied code by a function call to a newly created function that encapsulates the copied code [27, 54]. The use of design patterns is another option to avoid clones by better design [11, 12] although this approach can hardly

\textsuperscript{1}Trademark of Semantic Designs, Inc.
be automated. Likewise, one can develop code generators for highly repetitive code [39].

The challenge for the removal of clones is to find the right abstractions and the right level at which to remove clones (code, architecture, requirements).

4.4 Prevention and Compensation

The earlier clones are remedied, the easier it is likely to remove them. Lague et al. [62] propose to integrate clone detection in the normal development process. It can be used as preventive control where the code is checked continuously—for instance, at each check-in in the version control system or even on the fly while the code is edited. It can then be removed or documented.

A complementary integration is problem mining where the code currently under modification is searched in the rest of the system. The found segments of code can then be checked whether the change must be repeated in this segment for consistency.

Lague et al. assessed the benefits of integrating clone detection in normal development in a large postmortem case study and found many opportunities where their strategy could have helped [62]. It is interesting to note, that—contrary to their expectations—they observed a low rate of growth in the number of overall clones in the system, due to the fact that many clones were actually removed from the system.

Linked editing allows one to link two or more code clones persistently and to synchronize changes in one fragment with all other fragments automatically. It is an alternative to removing clones [85].

An open issue is which technique is better under which circumstances for which types of clones. Also, we need to find ways to rank clones according to their risks in order to direct measures to the most problematic clones.

5 Conclusions

This section summarizes the open issues of the subareas in software cloning presented in this paper.

One fundamental issue is that there is no clear consensus on what is a software clone. We should develop a general notion of redundancy, similarity, and cloning, and then identify more task-oriented categorizations of clones.

Concerning types of clones, we should look at alternative categorizations of clones that make sense (e.g., semantics, origins, risks, etc.). On the empirical side of clone categorizations, we should gather the statistical distribution of clone types in practice. Studying which strategies of removal and avoidance, risks of removal, potential damages, root causes, and other factors are associated with these categories would be worthwhile, too.

More empirical research is needed to reveal the root causes and main drivers for code cloning to broaden the insights of the existing studies. These studies should take industrial systems into account, too, as it is unclear to which extent these current observations can be attributed to the nature of open-source development.

In particular, empirical investigations of costs and benefits of clone removal are needed so that informed refactoring decisions can be made.

Unwanted clones should be avoided right from the start. But it is not yet clear what is the best integration of clone detection in the normal development process. In particular, what are the benefits and costs of such possible integrations and what are reliable cloning indicators to trigger refactoring actions?

If it is too late to avoid cloning and if existing clones cannot be removed, we should come up with methods and tools to manage these clones. This clone management must stop further spread of clones and help to make changes consistently.

The most elaborated field in software cloning is the automatic detection of clones. Yet, there is still room for improvement as identified in the quantitative and qualitative comparisons. Most helpful would be a ranking function that allows to present clone candidates in an order of relevance. This ranking function can be based on measures such as type, frequency, and length of clones but should also take into account the task driving the clone detection.

Clone detection is an enabling technology for many other maintenance areas. It may be used to detect aspects [16, 17], reduce the memory footprint of programs for small devices [20], detect plagiarism [82, 76, 29, 69, 38, 68, 36], and to compare versions or variants [86, 99, 34, 35, 88, 96, 97, 9, 31, 73]. I expect that clone detection will make many more contributions in related fields and in turn will learn from these other fields.

Currently, we are focusing very much on program text. However, the first approaches are underway to extend clone detection to other artifacts such as diagrams [23].

References


