2

Identifying and Removing Software Clones

Rainer Koschke

Universität Bremen, Germany

Summary. Ad-hoc reuse through copy-and-paste occurs frequently in practice affecting the evolvability of software. Researchers have investigated ways to locate and remove duplicated code. Empirical studies have explored the root causes and effects of duplicated code and the evolution of duplicated code. This chapter summarizes the state of the art in detecting, managing, and removing software redundancy. It describes consequences, pros and cons of copying and pasting code.

2.1 Introduction

A venerable and long-standing goal and ideal in software development is to avoid duplication and redundancy. Yet, in reality code duplication is a common habit. Several authors report on 7–23% code duplication [29, 291, 303]; in one extreme case even 59% [156].

Duplication and redundancy can increase the size of the code, make it hard to understand the many code variants, and cause maintenance headaches. The goal of avoiding redundancy has provided the impetus to investigations on software reuse, software refactoring, modularization, and parameterization. Even in the face of the ethic of avoiding redundancy, in practice software frequently contains many redundancies and duplications. For instance the technique of "code scavenging" is frequently used, and works by copying and then pasting code fragments, thereby creating so-called "clones" of duplicated or highly similar code. Redundancies can also occur in various other ways, including because of missed reuse opportunities, purposeful duplication because of efficiency concerns, and duplication through parallel or forked development threads.

Because redundancies frequently exist in code, methods for detecting and removing them from software are needed in many contexts. Over the past few decades, research on clone detection have contributed towards addressing the issue. Techniques for finding similar code and on removing duplication have been investigated in several specific areas such as software reverse engineering, plagiarism in student programs, copyright infringement investigation, software evolution analysis, code

T. Mens, S. Demeyer (eds.), *Software Evolution*.
DOI 10.1007/978-3-540-76440-3, © Springer 2008

compaction (e.g., for mobile devices), and design pattern discovery and extraction. Common to all these research areas are the problems involved in understanding the redundancies and finding similar code, either within a software system, between versions of a system, or between different systems.

Although this research has progressed over decades, only recently has the pace of activity in this area picked up such that significant research momentum could be established. This chapter summarizes the state of the art in detecting, managing, and removing software redundancy. It describes consequences, pros and cons of copying and pasting code.

Software clones are important aspects in software evolution. If a systems is to be evolved, its clones should be known in order to make consistent changes. Cloning is often a strategic means for evolution. For instance, copies can be made to create a playground for experimental feature evolution, where modifications are made in cloned code of a mature feature reducing the risk to break stable code. Once stable, the clone can replace its original. Often, cloning is the start of a new branch of evolution if the changes in the cloned code are not merged back to the main development branch. Clone detection techniques play an important role in software evolution research where attributes of the same code entity are observed over multiple versions. Here, we need to identify for an entity in one version the corresponding entity in the next version (known as *origin analysis* [568]). If refactoring (as for instance *renaming*) is applied between versions, the relation between entities of different versions is not always obvious. And last but not least, the evolution of clones can be studied to better understand the nature of cloning in practice.

2.2 Software Redundancy, Code Cloning, and Code Duplication

There are different forms of redundancy in software. Software comprises both programs and data. In the data base community, there is a clear notion of redundancy that has lead to various levels of normal forms. A similar theory does not yet exist for computer programs.

In computer programs, we can also have different types of redundancy. We should note that not every type of redundancy is harmful. For instance, programming languages use redundant declarations so that a compiler is able to check consistency between declarations and their uses. Also, at the architectural level, n-version programming is a strategy in which redundancy is purposefully and consciously used to implement reliable systems.

Sometimes *redundant* is used also in the sense of *superfluous* in the software engineering literature. For instance, Xie and Engler show that superfluous (they call them redundant) operations such as idempotent operations, assignments that are never read, dead code, conditional branches that are never taken, and redundant NULL-checks can pinpoint potential errors [550, 551].

Redundant code is also often misleadingly called *cloned* code in the literature—although that implies that one piece of code is derived from the other one in the original sense of this word. According to the Merriam-Webster dictionary, a *clone*

is one that appears to be a copy of an original form. It is a synonym to *duplicate*. Although cloning leads to redundant code, not every redundant code is a clone. There may be cases in which two code segments that are no copy of each other just happen to be similar or even identical by accident. Also, there may be redundant code that is semantically equivalent but has a completely different implementation.

There is no agreement in the research community on the exact notion of redundancy and cloning. Ira Baxter's definition of clones expresses this vagueness:

> Clones are segments of code that are similar according to some definition of similarity. —Ira Baxter, 2002

According to this definition, there can be different notions of similarity. They can be based on text, lexical or syntactic structure, or semantics. They can even be similar if they follow the same pattern, that is, the same building plan. Instances of design patterns and idioms are similar in that they follow a similar structure to implement a solution to a similar problem.

Semantic similarity relates to the observable behavior. A piece of code, *A*, is semantically similar to another piece of code, *B*, if *B* subsumes the functionality of *A*, in other words, they have "similar" pre and post conditions.

Unfortunately, detecting such semantic similarity is undecidable in general although it would be worthwhile as you can often estimate the number of developers of a large software system by the number of hash table or list implementations you find.

Another definition of cloning considers the program text: Two code fragments form a clone if their program text is similar. The two code fragments may or may not be equivalent semantically. These pieces are redundant because one fragment may need to be adjusted if the other one is changed. If the code fragments are executable code, their behavior is not necessarily equivalent or subsumed at the concrete level, but only at a more abstract level. For instance, two code pieces may be identical at the textual level including all variable names that occur within but the variable names are bound to different declarations in the different contexts. Then, the execution of the code changes different variables. Figure 2.1 shows two textually identical segments in the line range of 4–6 and 10–12, respectively. The semantic difference is that the first segment sets a global variable whereas the second one a local variable. The common abstract behavior of the two code segments is to iterate over a data structure and to increase a variable in each step.

Program-text similarity is most often the result of *copy&paste*; that is, the programmer selects a code fragment and copies it to another location. Sometimes, these programmers are forced to copy because of limitations of the programming language. In other cases, they intend to reuse code. Sometimes these clones are modified slightly to adapt them to their new environment or purpose.

Clearly, the definition of redundancy, similarity, and cloning in software is still an open issue. There is little consensus in this matter. A study by Walenstein et al. [532], for instance, reports on differences among different human raters for clone candidates. In this study, clones were to be identified that ought to be removed and Walenstein et al. gave guidelines towards clones worthwhile being removed. The

```
1  int sum = 0;
2
3  void foo(Iterator iter){
4    for (item = first(iter); has_more(iter); item = next(iter)){
5      sum = sum + value (item);
6    }
7  }
8  int bar(Iterator iter){
9    int sum = 0;
10   for (item = first(iter); has_more(iter); item = next(iter)){
11     sum = sum + value (item);
12   }
13 }
```

Fig. 2.1. Example of code clones

human raters of the clones proposed by automated tools did rarely agree upon what constitutes a clone worth to be removed. While the sources of inter-rater difference could be the insufficient similarity among clones or the appraisal of the need for removal, the study still highlights that there is no clear consensus yet, even for task-specific definitions of clones.

Another small study was performed at the Dagstuhl seminar 06301 "Duplication, Redundancy, and Similarity in Software" 2007. Cory Kapser elicited judgments and discussions from world experts regarding what characteristics define a code clone. Less than half of the clone candidates he 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.

2.3 Types of Clones

Program-text clones can be compared on the basis of the program text that has been copied. We can distinguish the following types of clones accordingly:

- **Type 1** is an exact copy without modifications (except for whitespace and comments).
- **Type 2** is a syntactically identical copy; only variable, type, or function identifiers have been changed.
- **Type 3** is a copy with further modifications; statements have been changed, added, or removed.

Baker further distinguishes so called parameterized clones [28], which are a subset of type-2 clones. Two code fragments A and B are a parameterized clone pair if there is a bijective mapping from A's identifiers onto B's identifiers that allows an identifier substitution in A resulting in A' and A' is a type-1 clone to B (and vice versa).

While type-1 and type-2 clones are precisely defined and form an equivalence relation, the definition of type-3 clones is inherently vague. Some researchers consider

Table 2.1. Classification by Balazinska et al. [33] ©[1999] IEEE

- difference in method attributes (`static`, `private`, `throws`, etc.)
- single-token difference in function body
  - further distinction into type of token:
  - called method
  - parameter type
  - literal
  - ...
- token-sequence difference in function body
  - one unit (expression or statement) differs in token sequence
  - two units
  - more than two units

two consecutive type-1 or type-2 clones together forming a type-3 clone if the gap in between is below a certain threshold of lines [29, 328]. Another precise definition could be based on a threshold for the Levenshtein Distance, that is, the number of deletions, insertions, or substitutions required to transform one string into another. There is no consensus on a suitable similarity measure for type-3 clones yet.

The above simple classification is still very rough. Balazinska et al. introduced a more refined classification for function clones [33] as described in Table 2.1. This classification makes sense for selecting a suitable strategy for clone removal. For instance, the design pattern *TemplateMethod* may be used to factor out differences in the types used in different code fragments or the design pattern *Strategy* can be used to factor out algorithmic differences [31, 32]. Furthermore Balazinska et al. argue that each class is associated with a different risk in clone removal.

Kapser et al.'s classification is the most elaborated classification to date [267, 265, 264] (cf. Table 2.2). The first level is a hint about the distance of clones. An argument can be made (although there is no empirical study on this hypothesis) that it is likely that clones between files are more problematic than within the same file as that it is more likely to overlook the former clones when it comes to consistent changes. The second decision distinguishes which syntactic units are copied. The third gives the degree of similarity and the fourth may be used to filter irrelevant or spurious clones.

2.4 The Root Causes for Code Clones

A recent ethnographic study by Kim and Notkin [277] has shed some light on why programmers copy and paste code. By observing programmers in their daily practice they identified the following reasons.

Sometimes programmers are simply forced to duplicate code because of limitations of the programming language being used. Analyzing these root causes in more detail could help to improve the language design.

Furthermore, programmers often delay code restructuring until they have copied and pasted several times. Only then, they are able to identify the variabilities of their

Table 2.2. Classification by Kapser et al. [265, 264] ©[2003] IEEE

1. At first level, distinguish clones within the same or across different files
2. then, according to type of region:
   - functions
   - declarations
   - macros
   - hybrids (in more than one of the above)
   - otherwise (among typedefs, variable declarations, function signatures)
3. then, degree of overlap or containment
4. then, according to type of code sequence:
   - initialization clones (first five lines)
   - finalization clones (last five lines)
   - loop clones (60% overlap of bodies)
   - switch and if (60% overlap of branches)
   - multiple conditions: several switch and if statements
   - partial conditions: branches of switch/if are similar

code to be factored out. Creating abstract generic solutions in advance often leads to unnecessarily flexible and hence needlessly complicated solutions. Moreover, the exact variabilities may be difficult to foresee. Hence, programmers tend to follow the idea of extreme programming in the small by not investing too much effort in speculative planning and anticipation.

Systems are modularized based on principles such as information hiding, minimizing coupling, and maximizing cohesion. In the end—at least for systems written in ordinary programming languages—the system is composed of a fixed set of modules. Ideally, if the system needs to be changed, only a very small number of modules must be adjusted. Yet, there are very different change scenarios and it is not unlikely that the chosen modularization forces a change to be repeated for many modules. The triggers for such changes are called *cross-cutting concerns* (see also Chapter 9). For instance, logging is typically a feature that must be implemented by most modules. Another example is parameter checking in defensive programming where every function must check its parameters before it fulfills its purpose [92]. Then copy&paste dependencies reflect important underlying design decisions, namely, cross-cutting concerns.

Another important root cause is that programmers often reuse the copied text as a template and then customize the template in the pasted context.

Kapser et al. have investigated clones in large systems [266]. 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. The assumption is that the copied code takes a separate evolu-

tion path independent of the original. In such a case, changes in the copy may be made that have no side effect on the original code.

Templating is used as a method to directly copy behavior of existing code but appropriate abstraction mechanisms are unavailable. It was also identified as a main driver for cloning in Kim and Notkin's case study [277]. Templating is often found when a reused library has a relatively fixed protocol (that is, a required order of using its interface items) which manifests as laying out the control flow of the interface items as a fixed pattern. For instance, the code in Fig. 2.1 uses a fixed iteration scheme for variable `iter`.

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. Time pressure, for instance, does not leave much time to search for the best long-term solution. Unavailable information on the impact of code changes leads programmers to create copies in which they make the required enhancement; such changes then are less likely to affect the original code negatively. Inadequate performance measures of programmers' productivity in the number of lines of code they produce neither invite programmers to avoid duplicates.

2.5 Consequences of Cloning

There are plausible arguments that code cloning increases maintenance effort. Changes must be made consistently multiple times if the code is redundant. Often it is not documented where code has been copied. Manual search for copied code is infeasible for large systems and automated clone detection is not perfect when changes are made to the copies (see Section 2.8). Furthermore during analysis, the same code must be read over and over again, then compared to the other code just to find out that this code has already been analyzed. Only if you make a detailed comparison, which can be difficult if there are subtle differences in the code or its environment, you can be sure that the code is indeed the same. This comparison can be fairly expensive. If the code would have been implemented only once in a function, this effort could have been avoided completely.

For these reasons, code cloning is number one on the stink parade of bad smells by Beck and Fowler [183]. But there are also counter arguments. In Kapser and Godfrey's study [266], code cloning is a purposeful implementation strategy which may make sense under certain circumstances (see Section 2.4).

Cordy makes a similar statement [128]. He argues that in the financial domain, cloning is the way in which designs are reused. Data processing programs and records across an organization often have very similar purposes, and, consequently, the data structures and programs to carry out these tasks are therefore very similar. Cloning becomes then a standard practice when authoring a new program. Opponents would argue that a better means would be to pursue systematic and organized reuse through software product lines.

Cordy also argues that the attempt to avoid cloning may lead to higher risks. Making changes to central data structures bears the risk to break existing applications and requires to run expensive regression tests. Instead programmers tend to copy the data structure if they want to restructure or add a different view and make the necessary changes in the copy. Even the argument that errors must be fixed in every copy does not count, he states. Errors would not necessarily be fixed in the original data structure because the many running applications may already rely on these errors, Cordy argues. On the other hand, repeated work, need for data migration, and risk of inconsistency of data are the price that needs to be paid following this strategy. The Y2K problem has shown how expensive and difficult it is to remedy systems that have suffered from massive decentralized use of data structures and algorithms.

While it is difficult to find arguments for type-1 and type-2 clones, one can more easily argue in favor of type-3 clones. It is not clear when you have type-3 clones whether the unifying solution would be easier to maintain than several copies with small changes. Generic solutions can become overly complicated. Maintainability can only be defined in a certain context with controlled parameters. That is, a less sophisticated programmer may be better off maintaining copied code than a highly parameterized piece of code. Moreover, there is a risk associated with removing code clones [128]. The removal requires deep semantic analyses and it is difficult to make any guarantees that the removal does not introduce errors. There may be even organizational reasons to copy code. Code cloning could, for instance, be used to disentangle development units [128].

The current debate lacks empirical studies on the costs and benefits of code cloning. There are very few empirical studies that explore the interrelationship of code cloning and maintainability. All of them focus on code cloning and errors as one (out of many) maintainability aspect.

Monden et al. [374] analyzed a large system consisting of about 2,000 modules written in 1 MLOC lines of Cobol code over a period of 20 years. They used a token-based clone detector (cf. Section 2.8.2) to find clones that were at least 30 lines long. They searched for 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. Monden et al. have not further analyzed why these maintainability factors correlate in such a way with code cloning.

Chou et al. [113] investigated the hypothesis that if a function, file, or directory has one error, it is more likely that is 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. They explain the correlation of bugs and copy-and-paste primarily by programmer ignorance, but they also note that—in addition to ignorance—the prevalence of copy-and-paste error clustering among different device drivers and versions suggests that programmers believe that “working” code is correct code. They note that if the copied

code is incorrect, or it is placed into a context it was not intended for, the assumption of goodness is violated.

Li et al. [328] 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 for the systems *Linux kernel*, *FreeBSD*, *Apache*, and *PostgreSQL*. The false positive rate of their technique is 73% on average, where on average 14% of the potential problems are still under analysis by the developers of the analyzed systems.

2.6 Clone Evolution

There are a few empirical studies on the evolution of clones, which describe some interesting observations. Antoniol et al. propose time series derived from clones over several releases of a system to monitor and predict the evolution of clones [14]. Their study for the data base system *mSQL* showed that their prediction of the average number of clones per function is fairly reliable. In another case study for the Linux kernel, they found that the scope of cloning is limited [15]. Only few clones can be found across subsystems; most clones are completely contained within a subsystem. In the subsystem *arch*, constituting the hardware architecture abstraction layer, newer hardware architectures tend to exhibit slightly higher clone rates. The explanation for this phenomenon is that newer modules are often derived from existing similar ones. The relative number of clones seems to be rather stable, that is, cloning does not occur in peaks. This last result was also reported by Godfrey and Tu who noticed that cloning is common and steady practice in the Linux kernel [205]. However, the cloning rate does increase steadily over time. Li et al. [328] observed for the Linux kernel in the period of 1994 to 2004 that the redundancy rate has increased from about 17% to about 22%. They observed a similar behavior for FreeBSD. Most of the growth of redundancy rate comes from a few modules, including *drivers* and *arch* in Linux and *sys* in FreeBSD. The percentage of copy-paste code increases more rapidly in those modules than in the entire software suite. They explain this observation by the fact that Linux supports more and more similar device drivers during this period.

Kim et al. analyzed the clone genealogy for two open-source Java systems using historical data from a version control system [278]. A clone genealogy forms a tree that shows how clones derive in time over multiple versions of a program from common ancestors. Beyond that, the genealogy contains information about the differences among siblings. Their study showed that many code clones exist in the system for only a short time. 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 clones, in particular those with a long lifetime that have changed consistently with other elements in the same group cannot easily be avoided because of limitations of the programming language.

One subproblem in clone evolution research is to track clones between versions. Duala-Ekoko and Robillard use a clone region descriptor [155] to discover a clone

of version n in version $n+1$. A clone region descriptor is an approximate location that is independent from specifications based on lines of source code, annotations, or other similarly fragile markers. Clone region descriptors capture the syntactic block nesting of code fragments. A block therein is characterized by its type (e.g., `for` or `while`), a string describing a distinguishing identifier for the block (the `anchor`), and a corroboration metric. The anchor of a loop, for instance, is the condition as string. If two fragments are syntactic siblings, their nesting and anchor are not sufficient to distinguish them. In such cases, the corroboration metric is used. It measures characteristics of the block such as cyclomatic complexity and fan-out of the block.

2.7 Clone Management

Clone management aims at identifying and organizing existing clones, controlling growth and dispersal of clones, and avoiding clones altogether. Lague et al. [303] and Giesecke [199] distinguish three main lines of clone management:

- *preventive* clone management (also known as *preventive control* [303]) comprises activities to avoid new clones
- *compensative* clone management (also known as *problem mining* [303]) 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

This section describes research in these three areas.

2.7.1 Corrective Clone Management: Clone Removal

If you do want to remove clones, there are several way to do so. There are even commercial tools such as *CloneDr*[1] by *Semantic Designs* to automatically detect and remove clones. Cloning and automatic abstraction and removal could even be a suitable implementation approach as hinted by Ira Baxter:

> Cloning can be a good strategy if you have the right tools in place. Let programmers copy and adjust, and then let tools factor out the differences with appropriate mechanisms. —Ira Baxter, 2002

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 [166, 287]. In more difficult cases, when the difference is not just in the variable names that occur in the copied code, one may be able to replace by macros if the programming languages comes with a preprocessor. A preprocessor offers textual transformations to handle more complicated replacements. If a preprocessor is available, one can also use conditional compilation. As the excessive use of macros and

[1] Trademark of Semantic Designs, Inc.

conditional compilation may introduce many new problems, the solution to the redundancy problem may be found at the design level. The use of design patterns is an option to avoid clones by better design [31, 32]. Yet, this approach requires much more human expertise and, hence, can be less automated. Last but not least, one can develop code generators for highly repetitive code.

In all approaches, it is a challenge to cut out the right abstractions and to come up with meaningful names of generated functions or macros. Moreover, it is usually difficult to check the preconditions for these proposed transformations—be they manual or automated–in order to assure that the transformation is semantic preserving.

2.7.2 Preventive and Compensative Clone Management

Rather than removing clones after the offense, it may be better to avoid them right from the beginning by integrating clone detection in the normal development process. Lague et al. identify two ways to integrate clone detection in normal development [303].

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—and the addition of a clone is reported for confirmation.

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.

Preventive control aims at avoiding code clones when they occur first whereas problem mining addresses circumstances in which cloning has been used for a while.

Lague et al. assessed the benefits of integrating clone detection in normal development by analyzing the three-year version history of a very large procedural telecommunication system [303]. In total, 89 millions of non-blank lines (including comments) were analyzed, for an average size of 14.83 million lines per version. The average number of functions per release was 187,000.

Problem mining is assessed by the number of functions changed that have clones that were not changed; that is, how often a modification was missed potentially. Preventive control is assessed by the number of functions added that were similar to existing functions; that is, the code that could have been saved.

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.

They conclude from their data that preventive control would help to lower the number of clones. Many clones disappeared only long after the day they came into existence. Early detection of clones could lead to taking this measure earlier.

They also found that problem mining could have provided programmers with a significant number of opportunities for correcting problems before end-user experienced them. The study indicates a potential for improving the software quality and customer satisfaction through an effective clone management strategy.

An alternative to clone removal is to live with clones consciously. Clones can be managed, linked, and changed simultaneously using linked editing as proposed

by Toomim et al. [504]. Linked editing is also used by Duala-Ekoko and Robillard. Linked editing allows one to link two or more code clones persistently. The differences and similarities are then analyzed, visualized, and recorded. If a change needs to be made, linked editing allows programmers to modify all linked elements simultaneously, or particular elements individually. That is, linked editing allows the programmer to edit all instances of a given clone at once, as if they were a single block of code. It overcomes some of the problems of duplicated code, namely, verbosity, tedious editing, lost clones, and unobservable consistency without requiring extra work from the programmer.

2.8 Clone Detection

While there is an ongoing debate as to whether remove clones, there is a consensus about the importance to at least detect them. Clone avoidance during normal development, as described in the previous section, as well as making sure that a change can be made consistently in the presence of clones requires to know where the clones are. Manual clone detection is infeasible for large systems, hence, automatic support is necessary.

Automated software clone detection is an active field of research. This section summarizes the research in this area. The techniques can be distinguished at the first level in the type of information their analysis is based on and at the second level in the used algorithm.

2.8.1 Textual Comparison

The approach by Rieger et al. compares whole lines to each other textually [156]. To increase performance, lines are partitioned using a hash function for strings. Only lines in the same partition are compared. The result is visualized as a dotplot, where each dot indicates a pair of cloned lines. Clones may be found as certain patterns in those dotplots visually. Consecutive lines can be summarized to larger cloned sequences automatically as uninterrupted diagonals or displaced diagonals in the dotplot.

Johnson [258, 259] uses the efficient string matching by Karp and Rabin [268, 269] based on fingerprints, that is, a hash code characterizing a string is used in the search.

Marcus et al. [345] compare only certain pieces of text, namely, identifiers using latent semantic indexing, a technique from information retrieval. Latent semantic analysis is a technique in natural language processing analyzing relationships between a set of documents and the terms they contain by producing a set of concepts related to the documents and terms. The idea here is to identify fragments in which similar names occur as potential clones.

2.8.2 Token Comparison

Baker's technique is also a line-based comparison. Instead of a string comparison, the token sequences of lines are compared efficiently through a suffix tree. A suffix tree for a string S is a tree whose edges are labeled with substrings of S such that each suffix of S corresponds to exactly one path from the tree's root to a leaf.

First, Baker's technique summarizes each token sequence for a whole line by a so called *functor* that abstracts of concrete values of identifiers and literals [29]. The functor characterizes this token sequence uniquely. Assigning functors can be viewed as a perfect hash function. Concrete values of identifiers and literals are captured as parameters to this functor. An encoding of these parameters abstracts from their concrete values but not from their order so that code fragments may be detected that differ only in systematic renaming of parameters. Two lines are clones if they match in their functors and parameter encoding.

The functors and their parameters are summarized in a suffix tree, a tree that represents all suffixes of the program in a compact fashion. A suffix tree can be built in time and space linear to the input length [356, 30]. Every branch in the suffix tree represents program suffixes with common beginnings, hence, cloned sequences.

Kamiya et al. increase recall for superficially different, yet equivalent sequences by normalizing the token sequences [263]. For instance, each single statement after the lexical patterns `if(...)`, `for(...)`, `while(...)`, and `do` and `else` in C++ is transformed to a compound block; e.g., `if (a) b = 2;` is transformed to `if (a) {b = 2;}`. Using this normalization, the if statement can be matched with the equivalent (with parameter replacement) code `if (x) {y = 2;}`.

Because syntax is not taken into account, the found clones may overlap different syntactic units, which cannot be replaced through functional abstraction. Either in a preprocessing [485, 127, 204] or post-processing [234] 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 [377] or a full-fledged syntax analysis [204].

2.8.3 Metric Comparison

Merlo et al. gather different metrics for code fragments and compare these metric vectors instead of comparing code directly [303, 291, 353, 289]. An allowable distance (for instance, Euclidean distance) for these metric vectors can be used as a hint for similar code. Specific metric-based techniques were also proposed for clones in web sites [151, 307].

2.8.4 Comparison of Abstract Syntax Trees

Baxter et al. partition subtrees of the abstract syntax tree (AST) of a program based on a hash function and then compare subtrees in the same partition through tree matching (allowing for some divergences) [47]. A similar approach was proposed

earlier by Yang [557] using dynamic programming to find differences between two versions of the same file.

Suffix trees—central to token-based techniques following Baker's idea—can also be used to detect sequences of identical AST nodes. In the approach by Koschke et al. [294], the AST nodes are serialized in preorder traversal, a suffix tree is created for these serialized AST nodes, and the resulting maximally long AST node sequences are then cut according to their syntactic region so that only syntactically closed sequences remain.

The idea of metrics to characterize code and to use these metrics to decide which code segments to compare can be adopted for ASTs as well. Jian et al. [257] characterize subtrees with numerical vectors in the Euclidean space $\Re$ and an efficient algorithm to cluster these vectors with respect to the Euclidean distance metric. Subtrees with vectors in one cluster are potential clones.

2.8.5 Comparison of Program Dependency Graphs

Textual as well as token-based techniques and syntax-based techniques depend upon the textual order of the program. If the textual order is changed, the copied code will not be found. Programmers modify the order of the statements in copied code, for instance, to camouflage plagiarism. Or they use code cloning as in the templating implementation strategy (see Section 2.4), where the basic skeleton of an algorithm is reused and then certain pieces are adjusted to the new context.

Yet, the order cannot be changed arbitrarily without changing the meaning of the program. All control and data dependencies must be maintained. A program dependency graph [175] is a representation of a program that represents only the control and data dependency among statements. This way program dependency graph abstract from the textual order. Clones may then be identified as isomorphic subgraphs in a program dependency graph [298, 286]. Because this problem is NP hard, the algorithms use approximative solutions.

2.8.6 Other Techniques

Leitao [324] 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. Walter et al. [531] and Li et al. [327] cast the search for similar fragments as a data mining problem. Statement sequences are summarized to item sets. An adapted data mining algorithm searches for frequent item sets.

2.9 Comparison of Clone Detection Algorithms

The abundance of clone detection techniques calls for a thorough comparison so that we know the strength and weaknesses of these techniques in order to make an

informed decision if we need to select a clone detection technique for a particular purpose.

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.

Bailey and Burd compared three clone and two plagiarism detectors [27]. Among the clone detectors were three of the techniques later evaluated by a subsequent study by Bellon and Koschke [55], namely, the techniques by Kamiya [263], Baxter [47], and Merlo [353]. For the latter technique, Bailey used an own re-implementation; the other tools were original. The plagiarism detectors were JPlag [420] and Moss [452].

The clone candidates of the techniques were validated by Bailey, and the accepted clone pairs formed an oracle against which the clone candidates were compared. Several metrics were proposed to measure various aspects of the found clones, such as scope (i.e., within the same file or across file boundaries), and the findings in terms of recall and precision.

The syntax-based technique by Baxter had the highest precision (100%) and the lowest recall (9%) in this experiment. Kamiya's technique had the highest recall and a precision comparable to the other techniques (72%). The re-implementation of Merlo's metric-based technique showed the least precision (63%).

Although the case study by Bailey and Burd showed interesting initial results, it was conducted on only one relatively small system (16 KLOC). However, because the size was limited, Bailey was able to validate all clone candidates.

A subsequent larger study was conducted by Bellon and Koschke [54, 55]. Their likewise quantitative comparison of clone detectors was conducted for 4 Java and 4 C systems in the range of totaling almost 850 KLOC. The participants and their clone detectors evaluated are listed in Table 2.3.

Table 2.4 summarizes the findings of Bellon and Koschke's study. Row *clone type* lists the type of clones the respective clone detector finds (for clone types, see Section 2.3). The next two rows qualify the tools in terms of their time and space consumption. The data is reported at an ordinal scale $--$, $-$, $+$, $++$ where $--$ is worst (the exact measures can be found in the paper to this study [54, 55]). Recall and precision are determined as in Bailey and Burd's study by comparing the clone

Table 2.3. Participating scientists

Participant	Tool	Comparison
Brenda S. Baker [29]	*Dup*	Token
Ira D. Baxter [47]	*CloneDr*	AST
Toshihiro Kamiya [263]	*CCFinder*	Token
Jens Krinke [298]	*Duplix*	PDG
Ettore Merlo [353]	*CLAN*	Function Metrics
Matthias Rieger [156]	*Duploc*	Text

Table 2.4. Results from the Bellon and Koschke study. Adapted from [54, 55] ©[2007] IEEE

	Baker	Baxter	Kamiya	Krinke	Merlo	Rieger
Clone type	1, 2	1, 2	1, 2, 3	3	1, 2, 3	1, 2, 3
Speed	++	−	+	−−	++	?
RAM	+	−	+	+	++	?
Recall	+	−	+	−	−	+
Precision	−	+	−	−	+	−

detectors' findings to a human oracle. The same ordinal scale is used to qualify the results; exact data are reported in the paper [54, 55].

Interestingly, Merlo's tool performed much better in this experiment than in the experiment by Bailey and Burd. However, the difference in precision of Merlo's approach in this comparison to the study by Bailey and Burd can be explained by the fact that Merlo compared not only metrics but also the tokens and their textual images to identify type-1 and type-2 clones in the study by Bellon and Koschke.

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. Rysselberghe and Demeyer [525] compared text-based [156, 438], token-based [29], and metric-based [353] clone detectors for refactoring. They compare these techniques in terms of suitability (can a candidate be manipulated by a refactoring tool?), relevance (is there a priority which of the matches should be refactored first?), confidence (can one solely rely on the results of the code cloning tool, or is manual inspection necessary?), and focus (does one have to concentrate on a single class or is it also possible to assess an entire project?). They assess these criteria qualitatively based on the clone candidates produced by the tools. Figure 2.2 summarizes their conclusions.

Bruntink et al. use clone detection to find cross-cutting concerns in C programs with homogeneous implementations [93]. In their case study, they used *CCFinder*—Kamiya's [263] tool evaluated in other case studies, too—one of the Bauhaus[2] clone detectors, namely *ccdiml*, which is a variation of Baxter's technique [47], and the PDG-based detector *PDG-DUP* by Komondoor [286]. The cross-cutting concerns they looked for were error handling, tracing, pre and post condition checking, and memory error handling. The study showed that the clone classes obtained by Bauhaus' *ccdiml* can provide the best match with the range checking, null-pointer

criterion	most suitable technique
suitability	metric-based
relevance	no difference
confidence	text-based
focus	no difference

Fig. 2.2. Assessment by Rysselberghe and Demeyer. Adapted from [525] ©[2004] IEEE

[2] `http://www.axivion.com`.

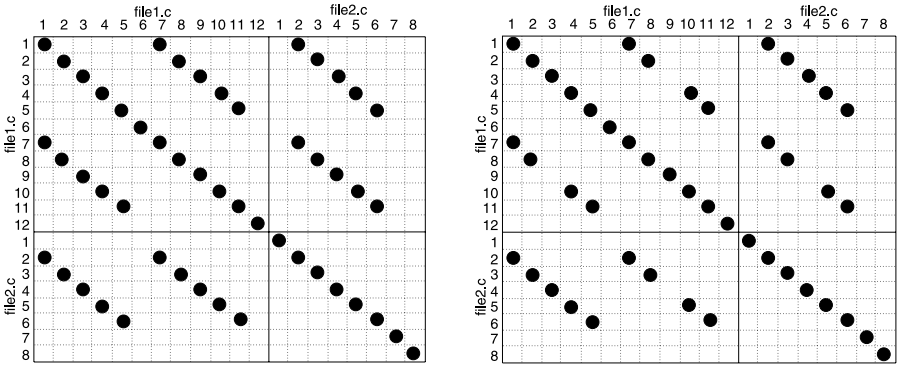
checking, and error handling concerns. Null-pointer checking and error handling can be found by *CCFinder* almost equally well. Tracing and memory error handling can best be found by *PDG-DUP*.

2.10 Clone Presentation

Because there is typically a huge amount of clones in large systems and these clones differ in various attributes (type, degree of similarity, length, etc.), presentation issues of clone information are critical. This huge information space must be made accessible to a human analyst. The analyst needs a holistic view that combines source code views and architectural views.

There have been several proposals for clone visualization. Scatter plots—also known as dot plots—are two-dimensional charts where all software units are listed on both axes [114, 156, 512] (cf. Fig. 2.3). There is a dot if two software units are similar. The granularity of software units may differ. It can range from single lines to functions to classes and files to packages and subsystems. Visual patterns of cloning may be observed by a human analyst. A problem with this approach is scalability for many software units and the order of the listed software units as this has an impact on the visual patterns. While there is a "natural" order for lines (i.e., lexical order) within a file, it is not clear how to order more coarse-grained units such as functions, files, and packages. Lexical order of their names is in most cases as arbitrary as random order.

Johnson [260] proposes Hasse diagrams for clone representation between sets of files so that one can better see whether code has been copied between files, which is possibly more critical than cloning within a file (cf. Fig. 2.4). A Hasse diagram (named after a German mathematician) is used to draw a partial order among sets as an acyclic graph. Directed arcs connect nodes that are related by the order relation and for which no other directed path exists.



(a) Three clones **(b)** Three clones, one modified line

Fig. 2.3. Dot plots [114, 156, 512]

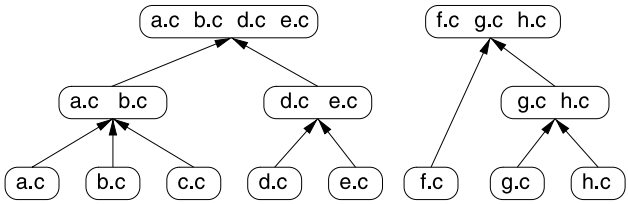


Fig. 2.4. Hasse diagram adapted from [260]

In Johnson's context, each match of a block of text identifies a range of characters (or lines) from two (or more) files. For each subset of files, one can total the number of characters that the matching process has discovered to match between the given set of files. A subset of files forms a node, if the files have non-zero matches. The inclusion between subsets of files yields the edges.

Rieger et al. [437] propose to use Michele Lanza's polymetric views[309] to visualize various aspects of clones in one view (cf. Fig. 2.5). A polymetric view is again based on the graph metaphor and representation where a node represents a software unit and an edge a cloning relation. Visually, additional information can be attached to the graph by the degrees of freedom for the position (X/Y in the two-dimensional space), color of nodes and edges, thickness of edges, breadth and width of nodes. Rieger et al. propose a fixed set of metric combinations to be mapped onto graphical aspects to present the clone information from different perspective for different tasks.

Beyond polymetric views, Rieger et al. [437] propose a variation of tree maps to show the degree of cloning along with the system decomposition (cf. Fig. 2.6). Tree maps display information about entities with a hierarchical relationship in a fixed space (for instance, the whole system on one screen) where the leaves of the hierarchy contain a metric to be visualized. Each inner node aggregates the metric values of its descendants. Each node is represented through a piece of the available space. The space of a descendent node is completely contained in the space of its ancestor.

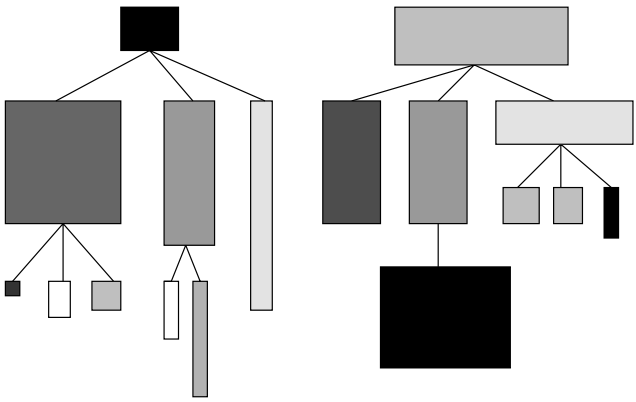


Fig. 2.5. Polymetric view adapted from [437]

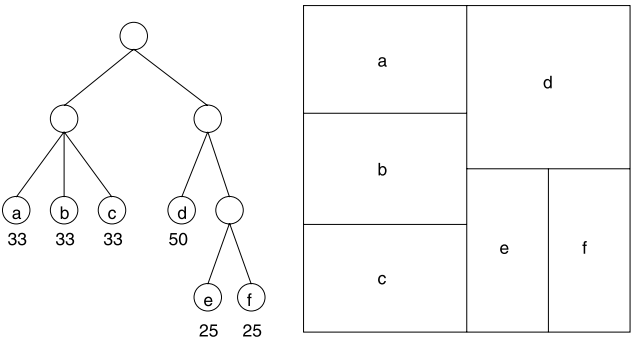


Fig. 2.6. A system decomposition whose leaves are annotated with the number of cloned lines of code and its corresponding tree map

There is no overlap in space for nodes that are not in an ancestor/descendant relation. This is how the hierarchy is presented. Essentially the hierarchy is projected onto the two dimensional space seen from the root of the hierarchy. In order to show the hierarchy clearly, the space of each node appears as rectangle where the direction of subdivision of nested nodes is alternated horizontally and vertically at each level. The space of each rectangle is proportional to the metric.

This visualization was originally proposed by Ben Shneiderman in the early 1990s to show space consumption of a hard disk with a hierarchical file system. While space is used very efficiently, problems arise when the hierarchy is deeply nested.

Another visualization was proposed by Wattenberg to highlight similar substrings in a string. The arc diagram has an arc connecting two equal substrings in a string where the breadth of the arc line covers all characters of the identical substrings. The diagram shows the overlapping of strings but becomes quickly unreadable if many arcs exist. Another disadvantage is that it shows only pairs but not classes of equal strings.

Tairas et al. [489] have created an Eclipse plugin to present clone information. One of their visualizations is the clone visualizer view, a window showing the distribution of clone classes among files (cf. Fig. 2.8). A bar in this view represents a source file, a stripe within a bar a cloned code segment, and its colors the set of clone classes the segment is member of.

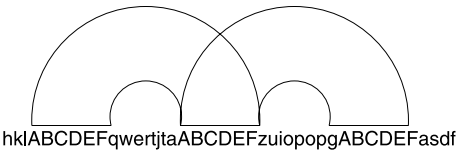


Fig. 2.7. Arc diagram adapted from [537]

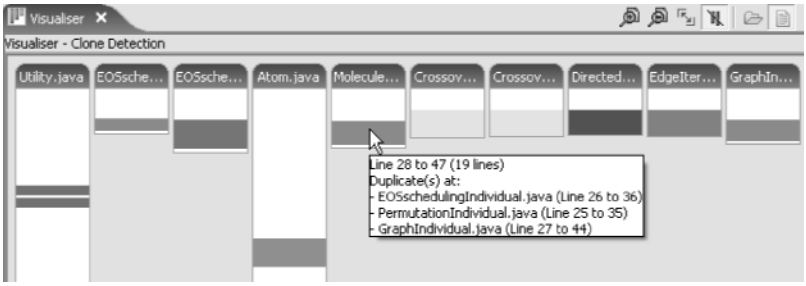


Fig. 2.8. Clones visualizer view in Eclipse adapted from [489]

2.11 Related Fields

Clone detection has applications in other fields and—vice versa—ideas from related fields can be reused for clone detection.

Bruntink et al. for instance, use clone detectors to search for code that could be factored out as aspects using an aspect-oriented language [92, 93]. They identify error handling, tracing, pre and post condition checking, and memory error handling. Although they used classic clone detectors that were not designed for this particular purpose, the clone detectors appeared to be helpful. Classic clone detectors try to find similar code—similar in terms of their program text. The implementations of an aspect, on the other hand, are often very heterogeneous and are similar only at a more semantic level. For instance, precondition checking tests each parameter of a function for certain criteria. At the implementation level, functions differ in the order and type of parameters so that checks are generally different in the program text.

The code compaction community tries to minimize the memory footprint of programs for small devices. They use very similar algorithms to identify redundant code that could be compressed [124].

The detection of plagiarism faces similar but even worse problems as clone detection [452, 420, 180, 343, 251, 337, 210]. In plagiarism cases, people try to camouflage their copy in order to make it more difficult to detect the plagiarism. In order to reuse classic clone detectors for plagiarism, we would need to reduce programs to a normal form for comparison. This normalization, on the other hand, could lead to false positives. Also in virus detection, code patterns significant for a particular hostile code need to be quickly identified in large code bases, where virus programmers try to vary the patterns.

Another application of clone detection is the comparison of versions or variants of software systems. While versions derive from each other, variants have a common ancestor. In both cases, they are very similar. In software evolution research, where information on software units is observed over time or versions, respectively, it is necessary to map the software entities of one version to those of the other version in order to carry over the information. This problem is called the *origin analysis* [508]. The same problem needs to be solved when two software variants are to be compared

or merged [237]. Relaying solely on names of these units for this analysis may be misleading if a refactoring like *renaming* has taken place [183]. Also, the refactoring *extract method* moves statements from one function to create a new function. Clone detection can help to establish a mapping between two versions or variants of a program. Several authors have used clone detection techniques or at least a code similarity measure to determine this mapping [509, 568, 206, 205, 524, 552, 553].

The difference of comparing versions or variants to detecting clones is that the task here is to map a code entity onto only one or at least a small set of candidates in the other system, the comparison is only between systems (clones within the same version or variant are irrelevant), cloning is the rule rather than the exception as the two versions or variants overlap to a very high degree, the focus is on the differences rather than the similarities, and the comparison should tolerate renaming and all refactorings that move entities around such as pull-up field, move method, etc.

2.12 Conclusions

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

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. Other research areas have similar difficulties in defining their fundamental terms. For instance, the architecture community debates the notion of *architecture* and the community of object-oriented programming the notion of *object*. To some extent, these fundamental terms define the field. So it is important to clarify them. It is difficult, for instance, to create benchmarks to evaluate automatic clone detectors if it is unclear what we consider a clone. It were very helpful if we could establish a theory of redundancy similar to normal forms in databases.

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 and investigate whether there are correlations among apparently orthogonal categories. 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.

Although the two empirical studies by Kim and Notkin as well as Kapser and Godfrey on the root causes and main drivers for code cloning are important first contributions, this area certainly requires more similar studies. Other potential reasons should be investigated, such as insufficient information on global change impact, badly organized reuse and development processes, questionable productivity measures (e.g., LOCs per day), time pressure, educational deficiencies, ignorance, or shortsightedness, intellectual challenges (e.g., generics), lack of professionalism/end-user programming by non experts, and organizational issues, e.g., distributed development and organizations.

Identifying the root causes would help us to fight the reasons, not just the symptoms, for instance, by giving feedback for programming language design.

Clearly, more empirical studies are required. 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. It would also be interesting to investigate what the degree of cloning tells about the organization or development process. For instance, a study by Nickell and Smith reports that the extreme programming projects in their organization produce significantly fewer clones [394, 533].

In particular, empirical investigations of costs and benefits of clone removal are needed so that informed refactoring decisions can be made. We currently do not have a clear picture of the relation of clone types to quality attributes. Most of what we report on the consequences of cloning is folklore rather than fact. We should expect that there is a relevance ranking of clone types for removal, that is, some clones should be removed, others are better left in certain circumstances. Moreover, we can expect that different types of clones are associated with different removal techniques in turn associated with different benefits, costs, and risks.

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.

Although various types of visualization to present clones have been proposed we have not fully explored all opportunities. There is a large body of research on information visualization in general and software visualization in particular that we have not yet explored for clone visualization. In order to understand which visualization works best for which purpose, we need more systematic empirical research. Clone representation is difficult due to the large and complex information space. We have various aspects that we need to master: the large amount of data, clone class membership, overlap and inclusion of clones, commonalities and differences among clones in the same class, degree of similarity, and other attributes such as length, type, frequency, and severity.

Clone detection overlaps with related fields, such as code compression or virus detection. The interesting questions here are “What can clone detection learn from other fields?” and “What can other fields learn from clone detection?”