Review of code clone articles

Wiebe Hordijk, María Laura Ponisio, Roel Wieringa
University of Twente, The Netherlands
hordijkwtb|m.l.ponisio|roelw@ewi.utwente.nl

Abstract

This report presents the results of a structured review of code clone literature. The aim of the review is to assemble a conceptual model of clone-related concepts which helps us to reason about clones. This conceptual model unifies clone concepts from a wide range of literature, so that findings about clones can be compared with each other. The conceptual model is work in progress; more research is needed to refine the concepts.

1 Introduction

Duplication of logic in source code is an important factor that is suspected to affect changeability. We want to investigate how design decisions affect duplication, and how duplication affects changeability. After some initial case investigations, we found that for a deeper research into the relationships between these variables, we needed a better understanding of the concepts behind code duplication. We also found some confusion of concepts in current literature about code duplication. To better understand the existing concepts and relationships behind code duplication, we have undertaken this structured review.

1.1 Problems for researchers and practitioners

Duplication of logic in source code is widely thought to negatively affect changeability [47, 48, 63]. We call this the *duplication hypothesis*. Recently some doubt has arisen about the truth of the duplication hypothesis in general. Our long-term research goal is to investigate the duplication hypothesis. We would like to refine it in terms of which kinds of clones affect changeability and which don't. To do this, we should analyze and compare the kinds of duplication mentioned in literature. We quickly found, however, that definitions of clones and related concepts in literature vary to an extent that makes comparison difficult. We illustrate this in section 2.

For practitioners it is difficult to put a clone detector to good use, because it is not clear which properties of clones relate to which quality attributes of the system. What tool should one use with which parameters to assess the system's maintainability? Should one use different parameters when searching for refactoring opportunities or when trying to prove plagiarism? Clone detection can be done for various reasons. These differences are not regarded by most articles. Note that use of code clone detection tools in practice is very limited, judging from the experience of the authors and from lack of experience reports. This may be caused by the lack of knowledge about which kinds of clones to detect for which purposes, which leads to low-quality clone detection results for the few practitioners who try these techniques.

The goal of this study is to create a common framework of concepts for reasoning about code clones and clone detectors, that enables us to compare and aggregate results about code duplication.

1.2 Structured review

We decided to attack the problems mentioned above using a structured review. This method allows us to collect and aggregate information from primary research in a structured way that is as repeatable as possible. Since our problems lie in the definitions used in literature, it is logical that we study literature to solve them. The approach is explained in section 3. Section 4 lists the results of the research identification and selection steps.

1.3 Framework

Because of the mentioned differences in concepts, findings about clones in literature are hard to compare and generalize. Therefore we feel the need for a framework in which these different concepts can be translated and compared. This framework should contain concepts, relations between those concepts, properties and possible values, such that concepts from all or most known code duplication literature can be stated in terms of the framework. This should enable us to compare findings of articles to each other. Based on our structured review we arrived at a framework into which all concepts from known literature can be translated; this framework is presented in section 5.

2 Code duplication literature hard to compare

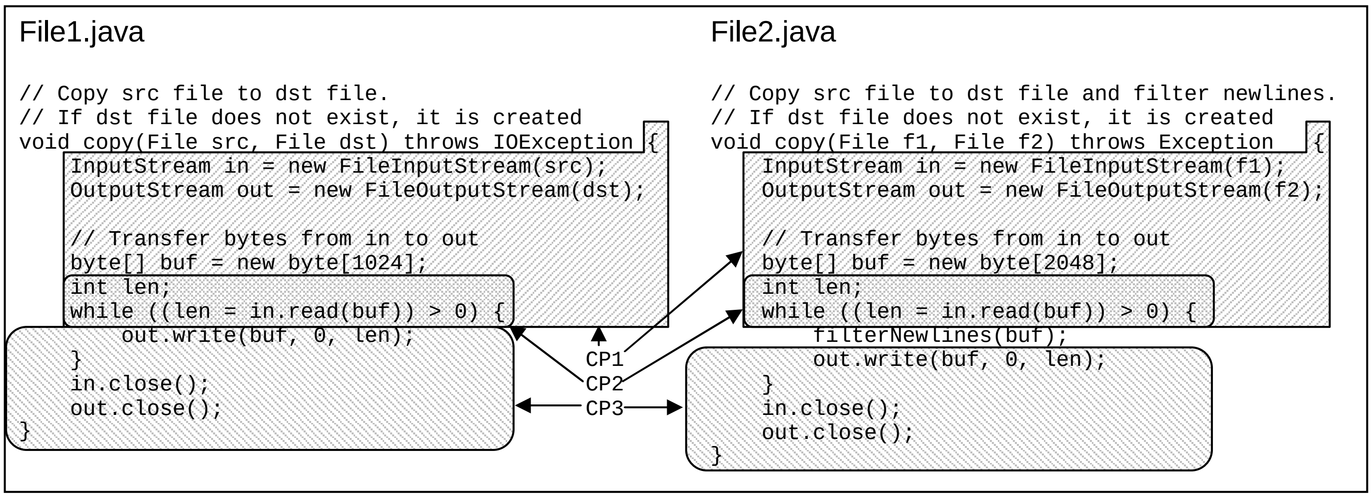
Code cloning has received ample attention from the software engineering community. The problem with the current body of code duplication research, is that it is difficult to compare approaches and findings to each other, because of several differences among them.

- Differences in their definitions and terminology;
- Different abstractions of the software they measure;
- Different goals with which tools were made or studies performed.

Comparing clones found using different definitions is difficult. For example, the comparison of clones detected by different clone detectors is problematic, for several reasons.

- They use different units of code: some tools use tokens, others use lines, yet others statements and predicates. These units are not convertible to each other, e.g. a line may contain multiple tokens, and tokens may cross line boundaries.
- A single clone detected by one tool may be detected as two separate clones in another. This makes it meaningless to compare numbers of clones detected by one tool to those of another.
- The tools have different ways of grouping detected clone pairs into clone sets.

The difficulty of comparing individual clones is illustrated in Figure 1, where a Java method has been copied and adapted from File1 to File2. The first comment line, parameter types, thrown exception, buffer size have been changed, and a line has been added to filter newlines. Two possible clone detection results from imaginary clone detectors have been highlighted. The square-edged clone detector has detected one clone pair (CP1), whose occurrences consist of tokens, starting with the opening accolade because the preceding token is different in the two code fragments, and ending just before the inserted line; it doesn't mind the differences in parameter names and constants. The round-edged clone detector is line based and only finds clones when at least two consecutive non-blank lines are exactly the same (CP2 and CP3). The square-edged clone detector does not find CP3 because it is below a clone size treshold. Now, which clone detector is right? Which has better recall or precision? Which of these clones should be refactored?



File1.java

```
// Copy src file to dst file.
// If dst file does not exist, it is created
void copy(File src, File dst) throws IOException {
    InputStream in = new FileInputStream(src);
    OutputStream out = new FileOutputStream(dst);

    // Transfer bytes from in to out
    byte[] buf = new byte[1024];
    int len;
    while ((len = in.read(buf)) > 0) {
        out.write(buf, 0, len);
    }
    in.close();
    out.close();
}
```

File2.java

```
// Copy src file to dst file and filter newlines.
// If dst file does not exist, it is created
void copy(File f1, File f2) throws Exception {
    InputStream in = new FileInputStream(f1);
    OutputStream out = new FileOutputStream(f2);

    // Transfer bytes from in to out
    byte[] buf = new byte[2048];
    int len;
    while ((len = in.read(buf)) > 0) {
        filterNewlines(buf);
        out.write(buf, 0, len);
    }
    in.close();
    out.close();
}
```

Figure 1 Clone pairs CP1, CP2 and CP3 detected according to different rules and definitions

Clone occurrences are often grouped into clone sets based on grouping rules, e.g. that every occurrence in a clone set should form a clone pair with every other in the same set. When clone sets of two clone detectors are compared to each other, things get more complicated than with clone pairs alone, as shown in Figure 2. The round-edged and square-edged clone detector both find clone 1, though the square-edged detector thinks it's a bit bigger. The round-edged detector finds a second clone which partly overlaps with the square-edged clone, but only in two files, and which occurs a second time in one of these files. Now, which detector has found more clones? Should we even regard clones as countable, or do they behave more like a mass noun such as water?

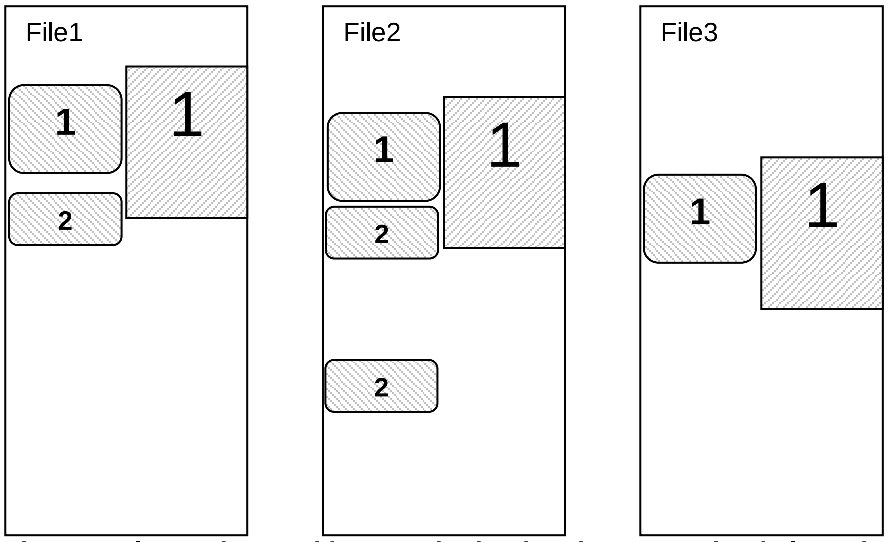


Figure 2 Clones detected by round-edged and square-edged clone detector in 3 source files

Whether it’s a problem that clones from different detectors are hard to compare depends on the goal with which we do the clone detection. If it is to better understand the structure of a software system, it may not matter. We however are trying to aggregate evidence from literature for the duplication hypothesis and to refine the duplication hypothesis by clone properties, so that we can reason about which kinds of clones are more harmful for changeability than others.

The problem is, then, that findings about different kinds of clones are hard to generalize from the literature, because of the mentioned differences in concepts. Therefore we feel the need for a framework in which these different concepts can be translated and compared. This framework should contain concepts, relations between those concepts, properties and possible values, such that concepts from all or most known code duplication literature can be translated to the framework. This should enable us to compare findings from different articles to each other.

3 Research method

This study is a structured review of the code clone literature. Information is gathered only from primary research, not from empirical observations. We have followed a method inspired by Kitchenham's general procedure for performing systematic reviews [77] because it maximizes the repeatability of the research effort. Even though the entire investigation is not completely repeatable, as human judgement is involved in interpreting articles, the method makes steps of the process as repeatable as possible. We used the following steps to arrive at our results.

1. Identification of research
2. Selection of primary studies
3. Text analysis
4. Synthesizing our framework from concepts and definitions
5. Checking our framework against the primary studies

Note that the primary studies are used in step 3 and in step 5. First we use them to build our framework, then we go back and check whether our framework still accomodates all the concepts of the primary sources. The last step is needed to enable others to check our findings.

3.1 Identification of research

We searched a number of literature sources with several search criteria, aimed at finding a set of articles with the most complete possible coverage of the field of code clones. The search criteria and results are documented in section 4.1.

3.2 Selection of primary studies

The following criteria were applied to the found sources for inclusion in this review.

- The article must be published in a journal or conference proceedings. This excludes drafts of articles and technical reports found on web sites of research groups.
- The article should have code clones as its primary subject. It can be about clone detectors, reasons why clones occur, or the impact of clones on quality attributes, but it must be about clones.
- The article should not be published before 1990. This boundary is chosen arbitrarily to limit the search for sources.

When a structured review is performed to aggregate quantitative data, for example in the field of medicine to aggregate the findings of different clinical tests for the effectiveness of a drug, this phase is followed by a quality assessment of the primary studies based on predefined criteria. In our case, this is not necessary, as our goal is to unify all views into the framework.

3.3 Text analysis

We analyzed the primary studies using text analysis techniques from conceptual modeling [110]. We searched the articles for pieces of text from which the meaning of concepts, the existence of relations between concepts, properties of concepts or the possible values of those properties can be deduced. We reported these quotes together with the framework deductions as raw data in a simple textual format. Some examples of quotes from articles and our deductions are given.

- *"Minimum clone length defines the minimum amount of lines present in a clone."* [16]
  - A *clone* has a property *length*
  - A *clone* consists of *lines of code*
  - The *minimum length* is a *similarity rule*
- *"Two lines of code are considered to be identical if they contain the same sequence of characters after removing comments and white space"* [7]
  - *Comments* and *white space* are *types of differences*
  - Allowing any *differences* of types *comment* and *white space* is a *similarity rule*

3.4 Synthesizing our framework

From the concepts identified in the previous step, we drew up a framework. The framework consists of two parts:

- A domain model of code clones, including concepts, definitions, attributes and possible values for those attributes, relations between concepts, and constraints.
- A generalized description of the processes needed to detect and report clones in software. These include rules e.g. used for detection of clone pairs. This can be used to compare clone detection tools on a conceptual level.

3.5 Checking our framework

We matched our framework against each of the primary articles. In the previous step, we have not only lumped together a number of definitions, but we have also added information, such as the relations and constraints, which is not written explicitly in any of the sources. Therefore this step is needed, in which we check whether our framework still matches all the information in the sources. Any mismatches and unclear parts are reported.

4 Article identification and selection

This section presents the results from steps 1, identification of research, and 2, selection of primary studies.

4.1 Identification of research

We used the following search engines with the specified search strings:

Searched in www.scopus.com on november 1, 2007 using search string:

```
      TITLE-ABS-KEY(+code +clones) AND LIMIT-TO(SUBJAREA,"COMP")
```

The limitation to the subject area of computing ("COMP") is necessary to filter out all articles about clones from biology and neighbouring fields. This has yielded 65 articles. Some of these clearly were not about code clones and were discarded. 38 articles remained. The two oldest articles are from 1996 and most are from 2007, showing that code duplication is a growing research field.

Some relevant articles about code clones apparently do not have "Comp" as one of their subject areas. We did another search where we included engineering ("ENG") and multi-disciplinary ("MULT") articles, and found 19 new relevant articles, which were added to the list.

We checked our list of articles for completeness by trying these other search criteria:

- Code duplication: 9 new articles, apparently not containing the term 'clone'.
- Code replication: no new articles. Apparently the term replication is used in cases where hardware is replicated to improve availability or where a piece of software replicates itself during runtime, such as viruses.
- Software clone: 1 new article, apparently not containing the word "code".
- Software duplication: 1 new article.
- Clone detection: no new articles.

We also searched the ACM guide to computing literature at http://portal.acm.org with the search string "code clone" and published after January 1990. This has yielded 8 more articles which could not be found in Scopus; those were added to our list. We did a cross-check to evaluate if starting with Scopus had been the right choice. It appeared that many results from the Scopus search did not occur in the ACM Portal search. While Scopus is less precise (it yields a lot of articles that are not relevant), it has higher recall, which is more important for a structured review.

We checked CiteSeer using a Google search with search string "site:citeseer.ist.psu.edu code clone". This gave 22 new articles. Some of these articles could not be found in Scopus or ACM Portal, even by searching on the full title. Some could be found, but do not have terms like clone or duplication in their title or abstract, which explains why they were not found in the previous searches. They use descriptions like "textual redundancy" [62] or "sections of code that are identical" [9]. Some have the word 'clone' in their title, but still did not come up in our earlier search [61] due to missing or incorrect meta information.

We have checked a sample of references in the selected articles to see if referenced articles about clones were missing in our list of articles, to check if our list of articles was complete. This has yielded 10 more papers. This indicates a potential validity problem as more papers may have been missed.

4.2 Selection of primary studies

We used the criteria specified in paragraph 3.2 to select and rate articles for inclusion in our review. The results of the searches are presented in Table 1. For each search, the references are given in the column 'Included' if they passed our criteria, else under 'Discarded'. The discarded papers are those that seemed to fit our criteria from looking at the title only, but were discarded after reading the paper itself. The row 'References' contains the papers that were not found in our searches but were added from bibliographies of other papers. The number of papers in each cell is given in parentheses.

Table 1 Results of the searches for literature

Database	**Search string**	**Included**	**Discarded**
Scopus	+code +clones COMP	(31) [1, 4-6, 11, 13, 18-20, 26, 37, 39-41, 44, 45, 54, 58, 59, 63, 67, 81, 84, 88, 92, 94, 96, 100, 102, 103, 111]	(7) [21, 30, 57, 64, 70, 89, 105]
Scopus	+code +clones ENG MULT	(15) [2, 14-17, 25, 29, 36, 46, 52, 56, 68, 76, 82, 108]	(4) [24, 34, 101, 107]
Scopus	+code +duplication	(6) [22, 42, 86, 87, 97, 109]	(3) [3, 10, 23]
Scopus	+software +clone		(1) [38]
Scopus	+software +duplication	(1) [8]	
ACM Portal	code clones	(5) [27, 55, 69, 79, 90]	(3) [53, 74, 99]
CiteSeer	code clone	(15) [7, 28, 31, 35, 50, 51, 61, 62, 66, 72, 83, 91, 93, 98, 104]	(7) [9, 12, 43, 65, 71, 75, 78]
References		(10) [32, 33, 48, 49, 60, 73, 80, 85, 95, 106]	
Totals		**(83)**	**(25)**

5 Framework

A framework that unifies the common concepts of code clone research is shown in a class diagram in Figure 3. The attributes and their possible values are described in the subsections.

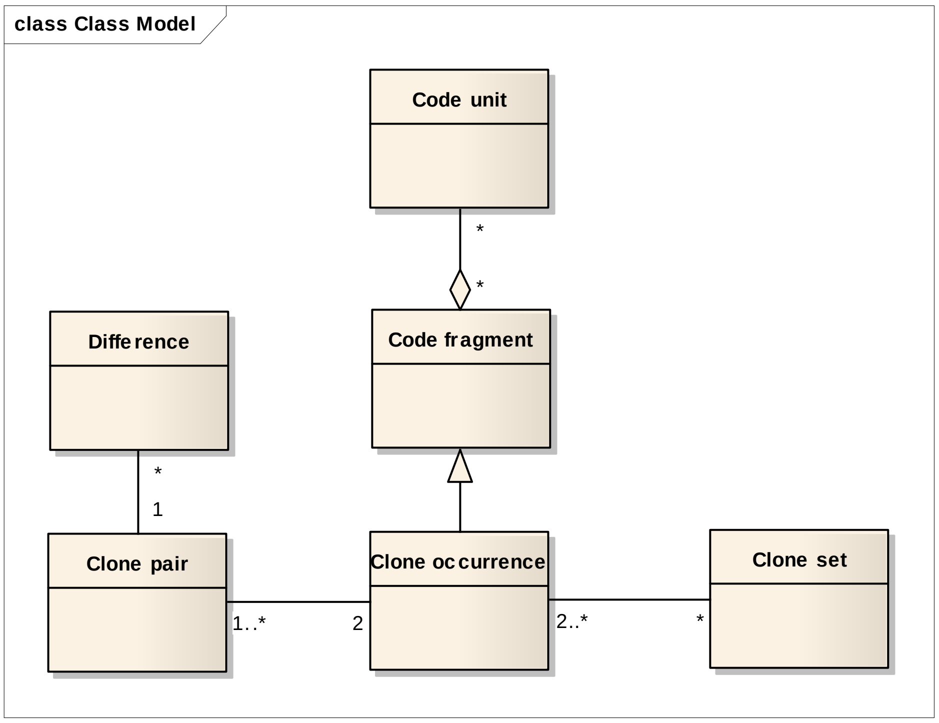
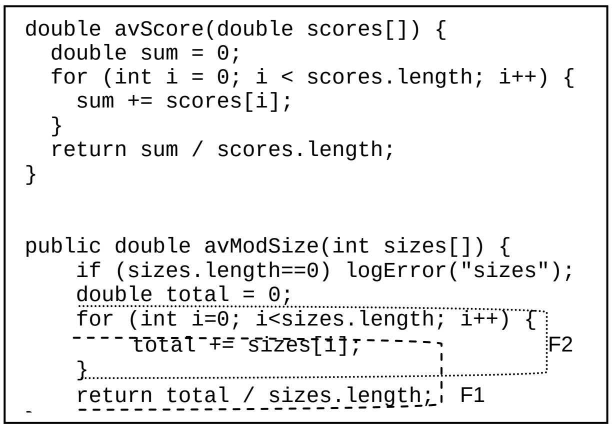


Figure 3 Code clone domain model

First, we give definitions of the concepts in Figure 3 and explain the relations. Then we enumerate and define their attributes.

5.1 Concepts and relations

A **code unit** is a small, atomic unit of source code. With atomic we mean that the internal structure of a code unit is considered unimportant. Code units cannot overlap with each other. Breaking the code into units is the first step of any clone detection effort. The type of units affects which detection rules can be used in subsequent steps of clone detection. Common examples of types of code units are tokens and lines.



```
double avScore(double scores[]) {
  double sum = 0;
  for (int i = 0; i < scores.length; i++) {
    sum += scores[i];
  }
  return sum / scores.length;
}


public double avModSize(int sizes[]) {
    if (sizes.length==0) logError("sizes");
    double total = 0;
    for (int i=0; i<sizes.length; i++) {
      total += sizes[i];                          F2
    }
    return total / sizes.length;  F1
```

Figure 4 Running example of a potential clone pair

A **code fragment** is a set of code units, e.g. a (part of a) method in Java. Most code fragments are sequences of code units, except for PDG slices which consist of statements with data or control dependences [79]. The concept of code fragment is important because clone detectors compare code fragments to find clone pairs.

Figure 4 shows a running example that we will use throughout this paper. It contains two methods that may or may not form or contain a clone pair, depending on the definitions used.

Code fragments can overlap with each other. Different clone detectors use different types of fragments. For example, detectors based on string comparison often allow arbitrary fragments [7, 42], which would allow F1 in the example, while parser-based detectors only allow sequences of AST subtrees that have the same parent [19] (F2 is an example). Most detectors that use metrics to compare fragments only use complete methods as fragments [80, 92]. Fragments may be represented in a different form, such as abstract syntax trees, but they can always also be represented as sequences of code units. Some clone detectors use multiple internal representations of fragments, for example, token strings for efficient comparison and AST subtrees for filtering.

A **clone occurrence** is a code fragment that is a member of a clone pair.

A **clone pair** is a pair of code fragments which have a certain similarity. The similarity is determined using a set of similarity rules. This set of rules may leave room for some differences between occurrences of a clone pair. The similarity rules form the most important aspect of a clone detection tool. Examples of similarity rules are “the fragments are identical” [7] and “the edit distance between the fragments is below a parameterized threshold” [68].

An alternative definition of clone pair focuses on the origin of the fragments: one must have been derived by copying the other [2]. Other definitions focus on the functionality (semantics) of the code: clone occurrences must ‘do’ the same thing, e.g. compute the same value from the same arguments [20]. Both these definitions have as a major drawback that they cannot be automatically measured from two fragments. Therefore all papers that mention such definitions use some sort of textual similarity as an objectification, and we have not included origin and semantics in the model.

A **difference** is a difference between the two occurrences in a clone pair. The detection rules of a clone detector determine which types of differences are allowed. Some examples are in 5.6. When a clone detector uses pre-processing of the code before clone detection, then this step may also cause certain types of differences to be allowed, such as white space, comments and pre-processing directives of the source language. In the latter case, the differences are not counted towards the size of the fragments. This has been modelled as a separate concept because it has significant impact on the refactorability of clones.

A **clone set** is a set of clone occurrences. Some clone detectors, not all, have a set of grouping rules to collect clone occurrences into sets. This is necessary if one wants to talk about the number of times a clone occurs in the system. Clone sets are usually formed such that each clone occurrence in the set forms a clone pair with every other clone occurrence in the same set. When differences between the clone occurrences in a clone pair are allowed, the rules for forming clone sets are non-trivial: occurrences (A, B) and occurrences (B, C) may form clone pairs, but (A, C) not; a clone occurrence may then belong to multiple clone sets. In other words, not all similarity rules form transitive relationships; similarity is not always equality.

5.2 Code unit properties

Type: The type of code unit.
Values:

- Character.
- Token. The types of tokens are further defined by tokenization rules.
- Statement.
- Predicate.
- Line of code.
- HTML tag (in web applications).

Position: The position of the code unit in the source code. Possible ways to describe the position depend on the type of the code unit. A line of code can be described by the file name and line number, for example. Any code units position can be described by file name, start position in the file and end position in the file.

Text: A textual representation of the content of the code unit as it appears in the source code.

Value: The value of the code unit as it will be used further in clone detection. For clone detectors that use textual identity, the value may be the same as the text, but for those that use tokenization, the value will be a token value, such as “CONST” for a token with text “42”.

Every code unit can be represented as a string of characters. We can consider strings as a canonical form of code unit to which all other types of code units can be transformed.

5.3 Code fragment properties

Type: The type of code fragment.
Values:

- Sequence. Any sequence of code units. In Figure 4, fragment F1 is a sequence of lines, but also a sequence of tokens.
- Block. A sequence of code units where the start and end are on the same level of nesting. An example in Figure 4 is F2.
- (Sequence of) AST subtree. A subtree of the Abstract Syntax Tree generated by parsing the code, or a sequence of AST subtrees under the same parent.
- Method. A whole method/function/procedure, including its header.
- Class.
- File.
- Module (also called Component)
- Program Dependence Graph (PDG) slice: a (possibly non-contiguous) set of code units with data or control dependences among them.
- Web page, in papers describing specific approaches for finding duplicated web pages. One paper [36] even uses a pair of web pages where one navigates to the other as fragment type.

Size: A count of code units in the fragment.

Dependences: A fragment may have dependences on its surrounding code, and vice versa. For example, a fragment may rely on a variable declaration outside the fragment; this can be detected automatically on a syntactic level.

Context type: The type of code in which the fragment occurs. How much of the environment to take into account is not defined, nor how to classify the context, but some values can be given.

- Front-end code: one paper [2] notices that code for user interfaces contains relatively many clones.
- Table initializations: when differences in variable names and literal values are allowed, many non-refactorable clones are found in table initializations [11].
- The tool Aries [52] lists several context types: Class, Static, Switch, Interface, Do, Synchronized, Method, For, Try, Constructor, If, While.
- Some papers [22, 58] focus on clones in library code, stating that those clones are a bigger problem than those in application code.

Position: The relative position of a fragment within its context. This can affect which refactorings are possible, and is especially interesting for aspect-oriented programming [26]. In a different context the position within the context is used for clone categorisation [50]. Values:

- Start of method.
- End of method.
- Middle of method: neither at the start nor the end.

Internal repetition: One paper [54] observes that a large amount of small sub-clones within a fragment, also called repetitive regions, indicates that the clone is not refcatorable. This occurs for example in table initialization code.

Other measures, such as cyclomatic complexity, can be computed when deemed interesting, if the nature of the fragment allows it.

5.4 Clone occurrence properties

Clone occurrences inherit the properties of code fragment. We have not identified additional properties.

5.5 Clone pair properties

Similarity: a detector-defined similarity measure.

Distance: The distance between the clone occurrences. One measure is 'dispersion' [52] which is -1 when classes have no common superclass in the code, 0 when occurrences are in the same class, and 1 when all occurrences are in direct subclasses.

Overlap: The degree to which the two code fragments overlap with each other. One formula is presented by Baker [11] as the number of code units the fragments have in common, divided by the number of code units in the union of the fragments. Many papers rank clones with overlapping fragments as 'false clones', meaning they cannot be refactored or have not been derived from copy-and-paste actions.

Origin: How the clone pair was created. This is considered important by many authors, but is not automatically measurable from source code. In some articles, a human programmer or researcher judges the origin of clone pairs by visual inspection, but without predefined rules [2]. The origin is important because in many articles only clone pairs that have originated from copying a fragment are considered 'real' clones.
Values:

- Copy and paste
- Typing from example
- Template: a specific piece of code is always programmed in a specific way, for example, retrieving data from a database table.
- Accident: the code 'looks the same', but does not 'do the same thing'.

Intention: The reason why the code has been cloned. This can be important for refactoring decisions because if there was a valid reason to duplicate the code, it may not be wise to eliminate the clone. We have collected some intentions mentioned in literature without being complete. Note that intentions are usually just mentioned as part of the motivation for studies and are not themselves subject to investigation.

- Lack of knowledge: the programmer was not aware that code was already present somewhere else and typed similar code.
- Saving initial effort: to avoid duplication, more initial effort would be needed because some abstraction must be programmed.
- Performance: the abstractions needed to avoid duplication would make the system less resource-efficient. This argument typically occurs in older papers.
- Maintaining architectural clarity: avoiding duplication would introduce dependencies between architecturally separate entities, or would violate architectural standards.
- Improving changeability: in some cases avoiding duplication would yield less changeable code because of the extra complexity needed.
- Risk of errors: to avoid duplication, one may need to change code that is known to work. The risk of introducing errors can be too big to take.
- Language limitations: some code clones cannot be avoided because the programming language lacks facilities for unification.
- Code ownership: one needs an adapted version of some code that one cannot change because it is used by another group.
- Increasing code size: some papers suggest that programmers copy code because their performance is assessed based on the amount of code they produce.

Conflict ratio: When systematic renaming of variables is an allowed type of difference between clone occurrences, one can allow a fraction of the variables to be renamed non-systematically [87]. This way one can find errors due to inconsistent renaming of variables.

5.6 Difference properties

Type: Differences are categorized based on the sort of editing operations one needs to perform to transform one clone occurrence into the other. Some types are generalizations of several more specific types of differences; we have listed specialisations in sub-lists.
Values:

- Different use of white space. Also described as code layout. The methods in Figure 4 have different formatting.
    - Different use of whitespace within lines. The line breaks must be the same.
- Comments are ignored by most clone detectors, although they too have to be maintained or else outdated comments will decrease changeability.
- Substitution of a token of one type by a token of a different type [17].
- (Systematic) Substitution of identifier: Substitution of an identifier by a different identifier. *Systematic* means: every use of identifier *x* in fragment 1 is replaced by identifier *y* in fragment 2. Also called p-matching [7].
    - Substitution of a variable name by a different variable name. The type of the variable (int, string etc.) is not taken into consideration. We consider as variables all kinds of variables including class fields, method parameters, array components and structure members. In Figure 4 there are systematic substitutions between ‘sum’ and ‘total’ and between ‘scores’ and ‘sizes’.
    - Substitution of a called method name.
    - Substitution of a literal value by a different literal value.
    - Substitution of a type by a different type.
- Substitution of a variable with an expression (not just a variable) of the same type. See [11] for examples.
- Insertion of a random code unit in one fragment. This is a broad category because it allows every kind of difference. In Figure 4 this is the first line of code in the second method.
- When the fragment type is ‘method’: a different method signature.
    - A different method name.
    - A different return type.
    - A different parameter type.
    - A difference in the set of thrown exceptions.
- Insertion of C preprocessor statements, Java ‘import’ statements etc. in one fragment.
- Insertion of block delimiters. Some clone detectors ignore block delimiters such as accolades.
- Insertion of common language constructs. Some clone detectors ignore common keywords like ‘if’, ‘for’, ‘while’ etc.
- Insertion of namespace qualifiers or package names.
- Insertion of template parameters (for languages that use generics).
- Insertion of array initialization lists (= {x, y, z, …}).
- Insertion of accessibility keywords (public, private, etc.).

Size: Some differences consist of code units. In those cases the size of a difference is defined as the number of code units it encompasses. For example, when random line insertions are allowed, the number of lines in one contiguous difference is its size. The random insertion in Figure 4 has one line.

5.7 Clone set properties

Population: The number of occurrences in the set.

Number of files: The number of different files in which fragments of the clone set occur.

Various other properties are defined on clone sets for specific purposes. These properties can be computed from properties of the clone occurrences in the set. We do not list all these properties.

6 Checking the famework

The last step in our review procedure is to review the selected articles to check if our framework can accommodate the ideas presented in them. In this section any discrepancies are motivated.

6.1 Framework rationale

In this paragraph we describe some interesting concepts from papers, which have prompted us to adjust our framework in some way. This gives the reader insight into our review process and the variety of concepts in the literature.

- PDG slices [79] are code fragments, eligible to be detected as clone occurrences, which do not necessarily consist of consecutive code units. They may be scattered all over the code. This has prompted us to redefine a code fragment from a sequence of code units to a set of code units.
- Repetitive regions [11] may be a good concept to add to the framework somehow, or show how it can be defined using the framework.
- The concept of 'system version' could be added to the concepts. However this is problematic because, depending on the point of view, concepts such as clone occurrence and clone set either belong to one version, or have an identity independent of the system version. Instead of trying to accommodate both views, we have kept our framework simple and added a 'source selection' step to the generalized detection process.
- Some papers aggregate individual clones into larger cloned 'regions' or 'patterns' or use them to compute clone relations between files. This is useful for code comprehension. We left this concept out of the framework because it diverges too much from our clone concepts.

6.2 Exceptions

In this paragraph we list concepts from papers which do not fit into our framework, and motivate why our framework cannot adapt to them.

- Some approaches use metrics, computed over fragments (often type 'method'), to compare fragments with each other when looking for clones. This fits in our framework, except that we cannot precisely define which differences are allowed by such a clone detector. For example, when cyclomatic complexity (CC) is used as a metric, and the difference between the CC of method 1 and the CC of method 2 cannot be greater than a threshold, then this rule allows virtually any kind of difference between the methods. Take into account that the detector uses several metrics values to determine similarity of fragments, and it is impossible to analyze which differences will be tolerated between clone occurrences.
- Some approaches exist where the similarity of code is measured through the similarity of the byte code (in the case of Java) resulting from compilation [12]. We have chosen to only consider papers in which duplication is defined as directly measurable on source code.
- One paper [35] presents a clone detector that uses a neural net, which is trained with a large number of clones and non-clones after which it can determine whether other fragments are clones or not. This approach falls outside our framework, because we cannot analyze which clones will be detected by the tool.
- One approach [99] uses natural language analysis on words in code and comments to determine similarity. This looks like a very interesting and unique approach that can give complementary information to other clone detectors. Its results, however, are too unpredictable for the approach to fit in our framework.

7 Conclusions

This report presents the results from a structured review of code duplication literature in the form of a conceptual model. We have determined that all concepts from the reviewed papers, with a few exceptions in section 6.2, can be stated in terms of the conceptual model. This conceptual model will aid us in future research where we intend to further investigate the duplication hypothesis.

A possible spin-off from the framework could be a standard for interoperability of clone detectors. It appears that most clone detectors can be described by the same reference model, in which information is exchanged between a certain number of components; these information flows can be described in terms of the framework, and generalized accordingly. This could result in a “clone interchange format” allowing practitioners for example to use the detection rules of one clone detector and the grouping rules of another, or to combine clones from two detectors in one visualisation.

Bibliography

This extended bibliography contains all papers included in and discarded from the review. The papers that were included are annotated with short descriptions after the dash. Discarded papers have a short explanation why they were discarded. Some references [47, 77, 110] are not papers included in the review, but methodological background about how we performed the review; those have not been annotated.

1. Adar, E. and M. Kim. *SoftGUESS: Visualization and exploration of code clones in context*. Proc. *International Conference on Software Engineering*. 2007. Minneapolis, MN. – Presents a clone visualization tool called SoftGUESS, using CCFinder for clone detection. Shows properties of clones within one version and across versions of the system.
2. Al-Ekram, R., et al. *Cloning by accident: An empirical study of source code cloning across software systems*. Proc. *International Symposium on Empirical Software Engineering*. 2005. – Measures the amount of cloning between open source systems, as an indicator for reuse of knowledge among open source projects in the same domain. Uses CCFinder for detection and CLICS for categorization. Found that front-end code contains more clones than back-end code.
3. Al-Ekram, R. and K. Kontogiannis. *Source code modularization using lattice of concept slices*. Proc. *European Conference on Software Maintenance and Reengineering, CSMR*. 2004. Tampere. – Discarded: duplication is not the main topic.
4. Antoniol, G., et al. *Modeling clones evolution through time series*. Proc. *Conference on Software Maintenance*. 2001. Florence. – Presents a method for monitoring and predicting clones evolution across versions of a system. Clones are identified using a metrics-based approach like Datrix. Time series is used to create a predictive model for future cloning in the system.
5. Antoniol, G., et al., *Analyzing cloning evolution in the Linux kernel.* Information and Software Technology, 2002. **44**(13): p. 755-765. – Studies the evolution of clones in Linux in terms of cloning ratio between subsystems.
6. Aversano, L., L. Cerulo, and M. Di Penta. *How clones are maintained: An empirical study*. Proc. *European Conference on Software Maintenance and Reengineering*. 2007. Amsterdam. – Investigates clone evolution patterns: consistent change, inconsistent change, independent evolution and late propagation. Makes a difference between changes for evolution (adaptive maintenance) and bug fixing. Found that late propagation occurs more often in clones with larger distance, and in small systems with less maintainers clones are maintained more consistently.
7. Baker, B.S., *A Program for Identifying Duplicated Code.* Computing Science and Statistics, 1992. **24**: p. 49-57. – Presents the clone detector Dup, explaining the similarity rules and the implementation of suffix trees and p-matching. Special attention for clones with internal repetition.
8. Baker, B.S. *On finding duplication and near-duplication in large software systems*. Proc. *2nd Working Conference on Reverse Engineering*. 1995. Toronto, Ont, Can. – Another presentation of Dup.
9. Baker, B.S., *Parameterized Pattern Matching: Algorithms and Applications.* Journal Computer System Science, 1996. **52**(1): p. 28-42. – Discarded: does not give new information.
10. Baker, B.S., *Parameterized duplication in strings: Algorithms and an application to software maintenance.* SIAM Journal on Computing, 1997. **26**(5): p. 1343-1362. – Discarded: does not give new information.
11. Baker, B.S., *Finding clones with dup: Analysis of an experiment.* IEEE Transactions on Software Engineering, 2007. **33**(9): p. 608-621. – Reaction on the experiment by Bellon (2002) where some clone detectors were compared on recall and precision w.r.t. a reference set of clones created by Bellon. Analyzes carefully which criteria Bellon used for 'oracling'. Indicates how Dup could be improved to score better on this particular experiment.
12. Baker, B.S. and U. Manber, *Deducing Similarities in Java Sources from Bytecodes.* Proc. of Usenix Annual Technical Conf., 1998: p. 179-190. – Discarded: concerns clones in compiled code rather than source code; this falls outside our scope.
13. Balazinska, M., et al. *Measuring clone based reengineering opportunities*. Proc. *International Software Metrics Symposium*. 1999. Boca Raton, FL, USA. – Investigates the use of clones as a basis for reengineering actions. Clones are categorized based on the number and types of differences between occurrences. Some categories are more refactorable than others.
14. Balazinska, M., et al. *Partial redesign of Java software systems based on clone analysis*. Proc. *6th Working Conference on Reverse Engineering*. 1999. Atlanta, GA, USA. – Presents an automated refactoring approach for clones, implemented in a tool called CloRT. Clones in sibling classes with as only differences systematically renamed variables are refactored to the superclass using a strategy pattern. Very precise about the allowed differences. An example is given in which the resulting code is much bigger than the original cloned code.
15. Balazinska, M., et al. *Advanced clone-analysis to support object-oriented system refactoring*. Proc. *Working Conference on Reverse Engineering*. 2000. – More elaborate presentation of CloRT.
16. Balint, M., T. Gîrba, and R. Marinescu. *How developers copy*. Proc. *14th IEEE International Conference on Program Comprehension*. 2006. Athens. – Combines clones with source control information (checkin time, developer) to detect patterns of how developers copy from each other. Uses Duploc for clone detection.

17. Basit, H.A. and S. Jarzabek. *Detecting higher-level similarity patterns in programs*. Proc. *European Software Engineering Conference*. 2005. – Presents a technique to detect file-level similarity patterns (sets of similar files) by first detecting basic clones (using definitions from CCFinder) and then using Frequent Itemset Mining to cluster the basic clone data into patterns. Implemented as CloneMiner.
18. Basit, H.A., D.C. Rajapakse, and S. Jarzabek. *Beyond templates: A study of clones in the STL and some general implications*. Proc. *27th International Conference on Software Engineering*. 2005. St. Louis, MO. – Shows that the STL contains clones that are hard to refactor using standard C++ templates, and shows how they can be refactored using a code generation technique XVCL.
19. Baxter, I.D., et al. *Clone detection using abstract syntax trees*. Proc. *Conference on Software Maintenance*. 1998. Bethesda, MD, USA. – Presents CloneDR, a clone detector based on Abstract Syntax Trees. Detects (sequences of) AST subtrees with small modifications as clones.
20. Bellon, S., et al., *Comparison and evaluation of clone detection tools.* IEEE Transactions on Software Engineering, 2007. **33**(9): p. 577-591. – Presents an experiment to compare some clone detectors on recall and precision w.r.t. a reference set. The reference set is created by taking a sample (2%) of the output of all clone detectors and judging (oracling) whether they are 'real' clones or not. Because clones detected by tools can partly overlap with reference clones, an algorithm is created to determine if they overlap enough to claim that a clone was correctly found by a tool.
21. Beyer, D., A. Noack, and C. Lewerentz. *Simple and Efficient Relational Querying of Software Structures*. Proc. *10th Working Conference on Reverse Engineering*. 2003. Victoria, BC. – Discarded: duplication is not the main topic, but only used as an example.
22. Black, A.P., N. Schärli, and S. Ducasse, *Applying traits to the smalltalk collection classes.* ACM SIGPLAN Notices, 2003. **38**(11): p. 47-64. – Presents traits, a Smalltalk language extension that enables sharing of functionality among classes that do not share the same inheritance hierarchy. Can be used to remove clones.
23. Bodík, R., R. Gupta, and M.L. Soffa, *Complete removal of redundant expressions.* ACM SIGPLAN Notices, 2004. **39**(4): p. 596-597. – Discarded: not about duplication but about runtime optimization of programs by compilers.
24. Bouktif, S., et al. *A novel approach to optimize clone refactoring activity*. Proc. *GECCO 2006 - Genetic and Evolutionary Computation Conference*. 2006. Seattle, WA. – Discarded: duplication is not the main topic; focus is on process improvement.
25. Bruntink, M., et al. *An evaluation of clone detection techniques for identifying crosscutting concerns*. Proc. *IEEE International Conference on Software Maintenance, ICSM*. 2004. Chicago, IL. – Investigates recall and precision of clone detectors w.r.t. a reference set of cross-cutting concerns (AOP). The reference set is created by manually searching for crosscutting concern code in part of a system. Depending on the type of concern, the recall varies; one can conclude that clone detection is a valuable aid in searching for some crosscutting concerns.
26. Bruntink, M., et al., *On the use of clone detection for identifying crosscutting concern code.* IEEE Transactions on Software Engineering, 2005. **31**(10): p. 804-818. – Extended version of Bruntink2004.
27. Burd, E. and J. Bailey. *Evaluating Clone Detection Tools for Use during Preventative Maintenance*. Proc. *2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*. 2002. – Compares 5 clone detectors on recall and precision by comparing their output. Criteria to evaluate clones are given based on differences between occurrences; goal is to judge whether clones were created by copy-paste.
28. Burd, E. and M. Munro. *Investigating the Maintenance Implications of the Replication of Code*. Proc. *International Conference on Software Maintenance*. 1997. – Investigates two cases of code duplication and their impact on maintenance. Clone detection is done manually.
29. Canfora, G., L. Cerulo, and M. Di Penta. *On the use of line co-change for identifying crosscutting concern code*. Proc. *IEEE International Conference on Software Maintenance, ICSM*. 2006. Philadelphia, PA. – Presents an approach for finding cross-cutting concerns (AOP) by combining clone detection (SimScan) and co-change data from release history.
30. Canfora, G., L. Cerulo, and M. Di Penta. *Identifying changed source code lines from version repositories*. Proc. *Fourth International Workshop on Mining Software Repositories, MSR* 2007. Minneapolis, MN. – Discarded: main topic is identifying changes in version repositories; duplication is only mentioned as an application of the proposed techniques.
31. Casazza, G., et al. *Identifying Clones in the Linux Kernel*. Proc. *International Workshop on Source Code Analysis and Manipulation*. 2001. – Evaluates the extent of cloning in the Linux kernel using a metrics-based clone detector.
32. Church, K.W. and J.I. Helfman, *Dotplot: A Program for Exploring Self-Similarity in Millions of Lines for Text and Code.* American Statistical Association, Institue for Mathematical Statistics and Interface Foundations of North America, 1993. **2**(2): p. 153-174. – Presents Dotplot, a clone detector based on similarity of lines; the user has to detect clones visually from the dotplot image.
33. Cordy, J.R., T.R. Dean, and N. Synytskyy. *Practical Language-Independent Detection of Near-Miss Clones*. Proc. *Conference of the Centre for Advanced Studies on Collaborative research*. 2004. Ontario, Canada. – Presents a clone detector for web pages, using an island grammar to find interesting fragments of source code, and then a lexer and line-based comparison.

34. Dagpinar, M. and J.H. Jahnke. *Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison*. Proc. *Working Conference on Reverse Engineering*. 2003. Victoria, BC. – Discarded: clones are only mentioned in the discussion.
35. Davey, N., et al., *The development of a software clone detector.* International Journal of Applied Software Technology, 1995. **1**(3-4): p. 219-36. – Presents a clone detector using a self organising neural net (SOM) to cluster feature vectors associated with program methods. Categorizes clones based on types of differences.
36. De Lucia, A., et al. *Reengineering web applications based on cloned pattern analysis*. Proc. *12th IEEE International Workshops on Program Comprehension*. 2004. Bari. – Presents an approach for reengineering Web Applications based on clone analysis that aims at identifying and generalizing static and dynamic pages and navigational patterns of a web application.
37. Di Lucca, G.A., M. Di Penta, and A.R. Fasolino. *An approach to identify duplicated web pages*. Proc. *IEEE Computer Society's International Computer Software and Applications Conference*. 2002. Oxford. – Presents an approach to identify duplicated web pages among and within web sites for aiding maintenance and detecting plagiarism.
38. Di Penta, M. *Evolution doctor: A framework to control software system evolution*. Proc. *European Conference on Software Maintenance and Reengineering, CSMR*. 2005. Manchester. – Discarded: position paper, no new information.
39. Di Penta, M., et al., *A language-independent software renovation framework.* Journal of Systems and Software, 2005. **77**(3): p. 225-240. – Presents a framework and toolkit to help reengineering. Duplication is one of the factors used to identify which parts to reengineer.
40. Duala-Ekoko, E. and M.P. Robillard. *Tracking code clones in evolving software*. Proc. *International Conference on Software Engineering*. 2007. Minneapolis, MN. – Presents an approach to track clones through multiple versions of a system by inserting clone region descriptors into the code. Also supports simultaneous editing of clones in Eclipse.
41. Ducasse, S., O. Nierstrasz, and M. Rieger, *On the effectiveness of clone detection by string matching.* Journal of Software Maintenance and Evolution, 2006. **18**(1): p. 37-58. – Assesses the quality of Duploc in terms of recall and precision, when extended with normalization of constants, variables and method names. Found that normalization of method names and allowing gap sizes larger than one line lead to a significant loss in precision and only minimal gain in recall, but normalization of constants and variables is needed to achieve decent recall.
42. Ducasse, S., M. Rieger, and S. Demeyer. *Language independent approach for detecting duplicated code*. Proc. *Conference on Software Maintenance*. 1999. Oxford, UK. – Presents Duploc, a purely text-based clone detector. Only language dependent parts are to remove comments and white space.
43. Ducasse, S.e., M. Rieger, and G. Golomingi. *Tool Support for Refactoring Duplicated OO Code*. Proc. *ECOOP'99 Workshop on Experiences in Object-Oriented Re-Engineering*. 1999. – Discarded: duplication is not the main topic; does not contain new information not included in other articles by same authors.
44. Fanta, R. and V. Rajlich, *Removing Clones from the Code.* Journal of Software Maintenance and Evolution, 1999. **11**(4): p. 223-243. – Presents a tool-set that removes clones from C++ programs, ensuring that preconditions hold (parameters). Clones have fragment type 'method' and can have systematically substituted variable names (parameters and local variables) and different names.
45. Fioravanti, F., G. Migliarese, and P. Nesi. *Reengineering analysis of object-oriented systems via duplication analysis*. Proc. *International Conference on Software Engineering*. 2001. Toronto, Ont. – Presents clone detector TREND. Based on line similarity of methods. Allows differences in white space, comments, pre-processor instructions, substitution of variable names (non-systematic) and random line insertions. Also presents duplication metrics on method, class and file level.
46. Flores, A. and M. Polo. *Dynamic component assessment on PvC environments*. Proc. *IEEE Symposium on Computers and Communications*. 2005. Murcia. – Uses clone detection on assertions written by a programmer about components.
47. Fowler, M., *Refactoring - Improving the Design of Existing Code*. Object Technology Series, ed. G. Booch, I. Jacobson, and J. Rumbaugh. 1999: Addison-Wesley.
48. Geiger, R., et al. *Relation of Code Clones and Change Couplings*. Proc. *Fundamental Approaches to Software Engineering*. 2006. – Quantifies the correlation between duplication and co-change on file level. Uses a specific definition of co-change: co-change occurs between two files if they are checked in at the same time by the same author with the same modification description. A statistical correlation was not found, but some pairs of files had much duplication and much co-change between them, providing weak evidence for duplication’s effect on co-change.
49. Gitchell, D. and N. Tran, *Sim: a utility for detecting similarity in computer programs.* SIGCSE Bull., 1999. **31**(1): p. 266-270. – Presents a clone detector called Sim which detects similarity of entire C programs, used for plagiarism detection among student assignments. Similarity is computed based on the edit distance of the programs parse trees.
50. Godfrey, M., et al. *Four Interesting Ways in Which History Can Teach Us About Software*. Proc. *International Workshop on Mining Software Repositories (MSR-04)*. 2004. Edinburgh, Scotland. – Position paper about a categorization of clones.
51. Godfrey, M.W. and L. Zou, *Using Origin Analysis to Detect Merging and Splitting of Source Code Entities.* IEEE Transactions on Software Engineering, 2005. **31**(2): p. 166-181. – Presents an approach to infer information about design changes from change history using origin analysis. Origin analysis uses clone

detection to find which code fragments originate from which fragments in previous versions. For clone detection, different detectors can be connected to the framework.

52. Higo, Y., et al. *Aries: Refactoring support environment based on code clone analysis*. Proc. *8th IASTED International Conference on Software Engineering and Applications*. 2004. Cambridge, MA. – Presents a set of clone metrics about dependencies, distance, size and population, that suggest which clones to refactor. Also presents a tool, Aries, to visualize this information.

53. Higo, Y., et al. *ARIES: refactoring support tool for code clone*. Proc. *3rd workshop on Software quality, ICSE*. 2005. St. Louis, Missouri. – Discarded: does not give new information when compared to other papers of same authors.

54. Higo, Y., et al., *Method and implementation for investigating code clones in a software system.* Information and Software Technology, 2007. **49**(9-10): p. 985-998. – Presents a visualization tool Gemini, based on CCFinder, to filter clones. Uses clone metrics on file and clone set level.

55. Higo, Y., et al. *On software maintenance process improvement based on code clone analysis*. Proc. *4th International Conference on Product Focused Software Process Improvement*. 2002. – Presents an extension to Gemini to improve the found clones by pruning them to refactorable fragment types.

56. Hill, R. and J. Rideout. *Automatic method completion*. Proc. *19th International Conference on Automated Software Engineering, ASE*. 2004. Linz. – Presents a technique to automatically compete a method that is being typed by a programmer when it can be detected that the method will be very similar to an already existing method. Similarity is based on feature vectors. From the examples it seems that the bigger part of the method must have been typed before detection can succeed.

57. Imai, T., Y. Kataoka, and T. Fukaya. *Evaluating software maintenance cost using functional redundancy metrics*. Proc. *IEEE Computer Society's International Computer Software and Applications Conference*. 2002. Oxford. – Discarded: duplication is not the main topic.

58. Jarzabek, S. and S. Li, *Unifying clones with a generative programming technique: A case study.* Journal of Software Maintenance and Evolution, 2006. **18**(4): p. 267-292. – Presents a technique to unify clones using XVCL (XML-based Variant Configuration Language), from which code can be generated. This enables to unify clones that are hard to refactor. Possible drawback is that an extra language is needed.

59. Jiang, L., et al. *DECKARD: Scalable and accurate tree-based detection of code clones*. Proc. *International Conference on Software Engineering*. 2007. Minneapolis, MN. – Presents Deckard, an AST-based clone detector. For efficiency, feature vectors are computed over AST subtrees, and similarity is measured as the euclidian distance of the feature vectors.

60. Johnson, J.H. *Identifying redundancy in source code using fingerprints*. Proc. *Conference of the Centre for Advanced Studies on Collaborative research: software engineering*. 1993. Toronto, Ontario, Canada. – Presents a clone detector that uses metrics (called fingerprints). Fragment type is sequence of lines; metrics are computed over all 50-line chunks of source code.

61. Johnson, J.H. *Substring Matching for Clone Detection and Change Tracking*. Proc. *International Conference on Software Maintenance (ICSM1 '94)*. 1994. – Presents a clone detector that uses metrics (called fingerprints). Fragment type is sequence of lines; metrics are computed over all 50-line chunks of source code.

62. Johnson, J.H. *Visualizing Textual Redundancy in Legacy Source*. Proc. *Conference of the Centre for Advanced Studies on Collaborative research*. 1994. – Takes the fingerprint matching a step further by combining clone data into higher-level matches on file level.

63. Kamiya, T., S. Kusumoto, and K. Inoue, *CCFinder: A multilinguistic token-based code clone detection system for large scale source code.* IEEE Transactions on Software Engineering, 2002. **28**(7): p. 654-670. – Presents CCFinder, a clone detector based on token sequences. Explains clone maximalization, p-matching and formation of clone sets. Differences are allowed through token identity and by not converting some source code items (e.g. comments) into tokens.

64. Kamiya, T., et al. *Maintenance support tools for JAVA programs: CCFinder and JAAT*. Proc. *International Conference on Software Engineering*. 2001. Toronto, Ont. – Discarded: position paper without new information.

65. Kapser, C. and M. Godfrey. *A taxonomy of clones in source code: The re-engineers most wanted list*. Proc. *2nd International Workshop on Detection of Software Clones*. 2003. – Discarded: position paper without new information.

66. Kapser, C. and M. Godfrey. *Toward a Taxonomy of Clones in Source Code: A Case Study*. Proc. *Evolution of Large-scale Industrial Software Applications (ELISA)*. 2003. Amsterdam. – Presents a categorization scheme for clones, based on the fragment type (method or block) and position (start, end or middle of method). Uses CCFinder and a self-made metrics-based clone detector for a case study.

67. Kapser, C. and M.W. Godfrey. *Aiding comprehension of cloning through categorization*. Proc. *International Workshop on Principles of Software Evolution (IWPSE)*. 2004. Kyoto. – Presents a clone categorization based on distance of the occurrences (same region, same file, same directory, different directory) and fragment type (method, block or macro).

68. Kapser, C. and M.W. Godfrey. *Improved tool support for the investigation of duplication in software*. Proc. *IEEE International Conference on Software Maintenance, ICSM*. 2005. Budapest. – Describes criteria for a clone detector, aimed at improving code comprehension. Presents a prototype of such a tool. Uses clone categorization by same authors (Kapser2004).

69. Kapser, C. and M.W. Godfrey. *"Cloning Considered Harmful" Considered Harmful*. Proc. *13th Working Conference on Reverse Engineering*. 2006. – Describes categories of clones of which some are harmful for changeability and some are not. No evidence is given.
70. Kapser, C.J. and M.W. Godfrey, *Supporting the analysis of clones in software systems: A case study.* Journal of Software Maintenance and Evolution, 2006. **18**(2): p. 61-82. – Discarded: does not add new information compared to other papers by same authors.
71. Kataoka, Y., et al. *Automated Support for Program Refactoring Using Invariants*. Proc. *International Conference on Software Maintenance*. 2001. – Discarded: duplication is not the main topic.
72. Kim, M. *Understanding and Aiding Code Evolution by Inferring Change Patterns*. Proc. *29th International Conference on Software Engineering*. 2007. – Position paper about using edit logs to infer change patterns.
73. Kim, M., et al. *An ethnographic study of copy and paste programming practices in OOPL*. Proc. *International Symposium on Empirical Software Engineering*. 2004. – Investigates copy&paste actions using an instrumented Eclipse editor. Found a number of valid reasons for copy&paste.
74. Kim, M. and D. Notkin. *Program element matching for multi-version program analyses*. Proc. *International workshop on Mining software repositories*. 2006. Shanghai, China. – Discarded: duplication is not the main topic.
75. Kim, M., D. Notkin, and D. Grossman. *Automatic Inference of Structural Changes for Matching across Program Versions*. Proc. *International Conference on Software Engineering*. 2007. – Discarded: duplication is not the main topic.
76. Kim, M., et al. *An empirical study of code clone genealogies*. Proc. *10th European Software Engineering Conference*. 2005. – Investigates the evolution of clone sets by using clone detection on multiple versions of a system using CCFinder. Found that a large proportion of clones are co-changed, though many only exist as clones during a small number of versions. They observe that many long-lived clones are not refactorable.
77. Kitchenham, B., *Procedures for Performing Systematic Reviews*. 2007, University of Durham, UK. -
78. Komondoor, R. and S. Horwitz, *Tool Demonstration: Finding Duplicated Code Using Program Dependences.* Lecture Notes in Computer Science, 2001. **2028**: p. 383-?? – Discarded: does not add new information.
79. Komondoor, R. and S. Horwitz. *Using Slicing to Identify Duplication in Source Code*. Proc. *8th International Symposium on Static Analysis*. 2001. – Presents a clone detector using Program Dependence Graphs (PDGs). This can detect non-contiguous fragments as clones, and fragments will always be refactorable as a whole.
80. Kontogiannis, K. *Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics*. Proc. *4th Working Conference on Reverse Engineering*. 1997. – Presents a clone detector based on metrics computed over AST representations of methods.
81. Kontogiannis, K.A., et al., *Pattern matching for clone and concept detection*, in *Reverse engineering*. 1996, Kluwer Academic Publishers. p. 77-108. - Presents two methods for clone detection, one based on text comparison and one on metrics, in the context of a tool set for maintenance of legacy systems.
82. Koschke, R., R. Falke, and P. Frenzel. *Clone detection using abstract syntax suffix trees*. Proc. *Working Conference on Reverse Engineering*. 2006. – Presents a clone detector based on AST subtrees, but using a suffix tree for efficiency. Allows for systematic substitution of variables.
83. Krinke, J., *Identifying Similar Code with Program Dependence Graphs.* Proc. Eigth Working Conference on Reverse Engineering, 2001: p. 301-309. – Presents a clone detector based on Program Dependence Graphs.
84. Lague, B., et al. *Assessing the benefits of incorporating function clone detection in a development process*. Proc. *Conference on Software Maintenance*. 1997. Bari, Italy. – Investigates clones in a system (using Datrix) and reasons as a thought experiment about what effect on maintainability two process changes would have had. Observes that with each release of the system, new clones are introduced, but also old clones disappear because of refactoring.
85. Lanubile, F. and T. Mallardo. *Finding function clones in Web applications*. Proc. *Seventh European Conference on Software Maintenance and Reengineering*. 2003. – Presents a clone detector for web applications, based on metrics, which categorizes the clones based in differences between the occurrences.
86. LaToza, T.D., G. Venolia, and R. DeLine. *Maintaining mental models: A study of developer work habits*. Proc. *International Conference on Software Engineering*. 2006. Shanghai. – Investigates developer work habits by surveys and interviews. Developers report that duplication is a big problem, but after further investigation it appeared that they meant co-change, not only because of duplication. Also presents a categorization of clones, based on how and why they were created, including opinions about how harmful they are.
87. Li, Z., et al., *CP-Miner: Finding copy-paste and related bugs in large-scale software code.* IEEE Transactions on Software Engineering, 2006. **32**(3): p. 176-192. – Presents clone detector CP-Miner, based on frequent itemset mining. Detects similar token strings with small gaps. Uses a p-matching algorithm (for systematic substitution of variables) that allows for a number of errors in the renaming, so that potential errors due to inconsistent renaming can be found.

88. Livieri, S., et al. *Analysis of the Linux kernel evolution using code clone coverage*. Proc. *Fourth International Workshop on Mining Software Repositories, MSR*. 2007. Minneapolis, MN. – Investigates cloning in 136 versions of the Linux kernel using D-CCFinder. Detects the clones between versions.
89. Livieri, S., et al. *Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder*. Proc. *International Conference on Software Engineering*. 2007. Minneapolis, MN. – Presents D-CCFinder, an extension of CCFinder that runs distributed on many computers to find clones in very large systems.
90. Lozano, A., M. Wermelinger, and B. Nuseibeh. *Evaluating the Harmfulness of Cloning: A Change Based Experiment*. Proc. *Fourth International Workshop on Mining Software Repositories*. 2007. – Measures co-change of clones on method level, using a tool for tracking clones across versions. Very preliminary; no clear conclusions.
91. Marcus, A. and J. Maletic. *Identification of High-Level Concept Clones in Source Code*. Proc. *16th IEEE international conference on Automated software engineering*. 2001. – Presents an approach to detect clones using the meaning of words in identifiers and comments. The technique is called Latent Semantic Indexing. This could complement other clone detection techniques.
92. Mayrand, J., C. Leblanc, and E.M. Merlo. *Experiment on the automatic detection of function clones in a software system using metrics*. Proc. *Conference on Software Maintenance*. 1996. Monterey, CA, USA. – Presents clone detector Datrix, based on a large set of metrics that are computed over methods. Each metric has a different threshold of maximal difference. The metrics are grouped into groups that are used for clone categorization, e.g. a clone that has a too big difference for a metric in the layout group, but matches according to the other groups, can fall into the 'DistinctLayout' category.
93. Mens, T., T. Tourwe, and F. Munoz. *Beyond the refactoring browser: Advanced tool support for software refactoring*. Proc. *International Workshop on Principles of Software Evolution IWPSE*. 2003. – Presents an approach to detect 'bad smells': indicators that part of a system should be refactored. One of these is duplication. A detector is presented based on AST subtrees.
94. Merlo, E., et al. *Investigating large software system evolution: The Linux kernel*. Proc. *IEEE Computer Society's International Computer Software and Applications Conference*. 2002. Oxford. – Presents metrics at release/system level to quantify similarities. Individual clones are detected using a metrics-based approach based on Kontogiannis 1996.
95. Monden, A., et al. *Software quality analysis by code clones in industrial legacy software*. Proc. *Eighth IEEE Symposium on Software Metrics*. 2002. – Measures the correlation between duplication and revision number (and numbers of errors). Modules with more duplication have higher revision numbers; this is taken as an indicator for lower changeability. Modules with more duplication have less errors per line, which is expected because they need more lines than an equivalent non-duplicated module. Interestingly, modules with very large clones have more errors per line.
96. Rajapakse, D.C. and S. Jarzabek. *Using server pages to unify clones in web applications: A trade-off analysis*. Proc. *International Conference on Software Engineering*. 2007. Minneapolis, MN. – Case study in which a web application was refactored using the Server page pattern. Discusses the advantages and disadvantages. To find clones, CCFinder is used.
97. Rieger, M., S. Ducasse, and M. Lanza. *Insights into system-wide code duplication*. Proc. *Working Conference on Reverse Engineering, WCRE*. 2004. Delft. – Presents a number of visualizations of clones, to enable programmers to see a number of metrics about a number of clones in an overview.
98. Rysselberghe, F.V. and S. Demeyer. *Evaluating Clone Detection Techniques*. Proc. *International Workshop on Evolution of Large Scale Industrial Software Applications*. 2003. – Evaluates clone detectors qualitatively with respect to where to use them in the maintenance process. Compares line matching, metrics and p-matching.
99. Shepherd, D., L. Pollock, and K. Vijay-Shanker. *Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task*. Proc. *7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2007. San Diego, California, USA. – Discarded: duplication is not the main topic.
100. Tairas, R. *Clone detection and refactoring*. Proc. *Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*. 2006. Portland, OR. – Position paper about research that should lead to tool support from clone detection to refactoring.
101. Tairas, R. and J. Gray. *Phoenix-based clone detection using suffix trees*. Proc. *44th annual Southeast regional conference*. 2006. Melbourne, Florida. – Presents a clone detector that compares AST subtrees using a suffix tree. Implemented in the Microsoft Phoenix framework.
102. Tairas, R., J. Gray, and I. Baxter. *Visualization of clone detection results*. Proc. *OOPSLA Workshop on Eclipse Technology eXchange, ETX*. 2006. Portland, OR. – Presents a tool that visualizes clones in Eclipse.
103. Tonella, P., et al., *Reverse engineering 4.7 million lines of code.* Software - Practice and Experience, 2000. **30**(2): p. 129-150. – Experience report about a large reengineering effort, part of which was clone detection and elimination.
104. Toomim, M., A. Begel, and S.L. Graham. *Managing Duplicated Code with Linked Editing*. Proc. *Symposium on Visual Languages - Human Centric Computing, VLHCC*. 2004. – Presents a tool for linked editing. A programmer can indicate that two fragments are clones. Then when one fragment is edited, the tool makes the same changes to the other fragment.

105. Uchida, S., et al., *Software analysis by code clones in open source software.* Journal of Computer Information Systems, 2005. **45**(3): p. 1-11. – Discarded: does not contain new information compared to papers by same authors.
106. Ueda, Y., et al. *On detection of gapped code clones using gap locations*. Proc. *Software Engineering Conference, 2002. Ninth Asia-Pacific*. 2002. – Presents an approach to infer clones with random differences (gapped clones) up to a size threshold from basic clones without such differences (detected by CCFinder). Uses a visualization so that users can manually identify gapped clones.
107. Wahler, V., et al. *Clone detection in source code by frequent itemset techniques*. Proc. *Fourth IEEE International Workshop on Source Code Analysis and Manipulation*. 2004. – Presents a clone detector based on frequent itemset mining over an XML representation of the parse tree of the source code.
108. Walenstein, A., et al. *Problems creating task-relevant clone detection reference data*. Proc. *10th Working Conference on Reverse Engineering*. 2003. – Investigates human judgements about clones. To create a reference set for evaluating clone detectors, clones were judged by three researchers. The paper discusses the large amount of disagreement between the judges and argues that this forms a generalizability problem for all clone detector evaluations.
109. Wettel, R. and R. Marinescu. *Archeology of code duplication: Recovering duplication chains from small duplication fragments*. Proc. *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC*. 2005. Timisoara. – Presents a clone detector that detects clones with gaps up to a maximum gap size.
110. Wieringa, R.J., *Requirements Engineering: Frameworks for Understanding*. 1996: Wiley.
111. Yamamoto, T., et al., *Similarity of software system and its measurement tool SMMT.* Systems and Computers in Japan, 2007. **38**(6): p. 91-99. – Presents system-level duplication metrics. Clones are detected using CCFinder.