Do Code Clones Matter?

Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, Stefan Wagner
Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{juergens,deissenb,hummelb,wagnerst}@in.tum.de

Abstract

Code cloning is not only assumed to inflate maintenance costs but also considered defect-prone as inconsistent changes to code duplicates can lead to unexpected behavior. Consequently, the identification of duplicated code, clone detection, has been a very active area of research in recent years. Up to now, however, no substantial investigation of the consequences of code cloning on program correctness has been carried out. To remedy this shortcoming, this paper presents the results of a large-scale case study that was undertaken to find out if inconsistent changes to cloned code can represent faults. For the analyzed commercial and open source systems we not only found that inconsistent changes to clones are very frequent but also identified a significant number of faults induced by such changes. The clone detection tool used in the case study implements a novel algorithm for the detection of inconsistent clones. It is available as open source to enable other researchers to use it as basis for further investigations.

1. Clones & correctness

Research in software maintenance has shown that many programs contain a significant amount of duplicated (cloned) code. Such cloned code is considered harmful for two reasons: (1) multiple, possibly unnecessary, duplicates of code increase maintenance costs and, (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior [19, 28]. While clone detection has been a very active area of research in recent years, up to now, there is no thorough understanding of the degree of harmfulness of code cloning. In fact, some researchers even started to doubt the harmfulness of cloning at all [16].

To shed light on the situation, we investigated the effects of code cloning on program correctness. It is important to understand, that clones do not directly cause faults but inconsistent changes to clones can lead to unexpected program behavior. A particularly dangerous type of change to cloned code is the *inconsistent bug fix*. If a fault was found in cloned code but not fixed in *all* clone instances, the system is likely to still exhibit the incorrect behavior. To illustrate this, Fig. 1 shows an example, where a missing null-check was retrofitted in only one clone instance.

This paper presents the results of a large-scale case study that was undertaken to find out (1) if clones are changed inconsistently, (2) if these inconsistencies are introduced intentionally and, (3) if unintentional inconsistencies can represent faults. In this case study we analyzed three commercial systems written in C#, one written in Cobol and one open-source system written in Java. To conduct the study we developed a novel detection algorithm that enables us to detect inconsistent clones. We *manually* inspected about 900 clone groups to handle the inevitable false positives and discussed each of the over 700 inconsistent clone groups with the *developers* of the respective systems to determine if the inconsistencies are intentional and if they represent faults. Altogether, around 1800 individual clone group assessments were manually performed in the course of the case study. The study lead to the identification of 107 faults that have been confirmed by the systems' developers.

Research Problem Although most previous work agrees that code cloning poses a problem for software maintenance, "there is little information available concerning the impacts of code clones on software quality" [28]. As the consequences of code cloning on program correctness, in particular, are not fully understood today, it remains unclear how *harmful* code clones really are. We consider the absence of a thorough understanding of code cloning precarious for software engineering research, education and practice.

Contribution The contribution of this paper is twofold. First, we extend the existing empirical knowledge by a case study that demonstrates that clones get changed inconsistently and that such changes can represent faults. Second, we present a novel suffix-tree based algorithm for the detection of inconsistent clones. In contrast to other algorithms for the detection of inconsistent clones, our tool suite is made available for other researchers as open source.

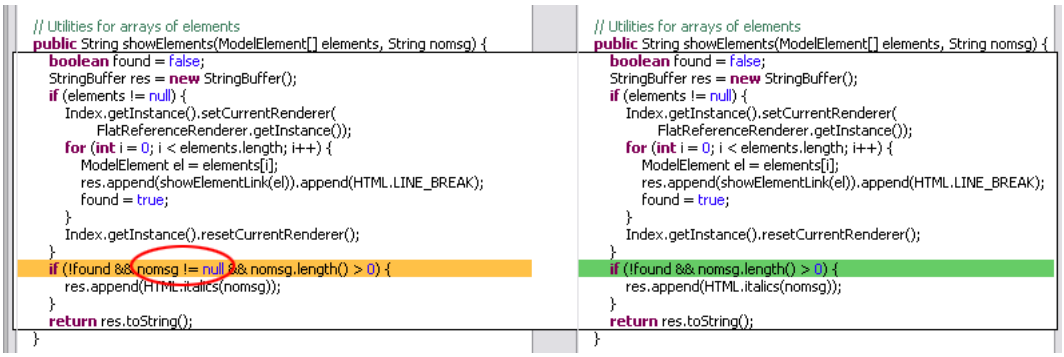

Figure 1. Missing null check on right side can cause exception (Sysiphus).

2. Terms and definitions

The literature provides a wide variety of different definitions of clones and clone related terms [19, 28]. To avoid ambiguity, we describe the terms as used in this paper.

Code is interpreted as a sequence of *units*, which for example could be characters, normalized statements, or lines. The reason to allow normalization of units at this stage, is that often pieces of code are considered equal even despite differences in comments or naming, which can be leveled by the normalization. An *exact clone* is then a (consecutive) substring of the code that appears at least twice in the (normalized) code. Thus our definition of a clone is purely syntactical, but catches exactly the idea of *copy&paste*, while allowing simple changes, such as renaming, due to normalization. An *exact clone group* is a set of at least two exact clones that appear at different positions.

To capture the notion of non-identical clones, we roughly follow the definitions of a gapped or type 3 clone given in [19, 28]. A substring s of the code is called an *inconsistent clone*, if there is another substring t of the code such that their edit distance is below a given threshold and that t has no significant overlap with s. The edit distance is a metric that counts the number of edit operations (insertion, removal, or change of a single unit) needed to transform one sequence into the other. Obviously, this definition is slightly vague, as it depends on the threshold chosen and the meaning of a "significant overlap". However, it captures our intuitive understanding of an inconsistent clone as used in this paper. Examples are shown in Figs. 1 and 7. By *clone* we denote both exact and inconsistent clones.

A *clone group* can be viewed as a *connected* graph, where each node is a substring, and edges are drawn between substrings that are clones of each other. If at least one pair of inconsistent clones is in the group, it is called an *inconsistent clone group*. We could also have required all clones in a clone group to be clones of each other, but often these slightly larger clone groups created by our definition reveal interesting relationships in the code.

For a thorough discussion of the consequences of inconsistent clones, we define that a *failure* is an incorrect output of a software visible to the user and that a *fault* is the cause of a potential failure inside the code. *Defects* are the superset of faults and failures.

3. Related work

A substantial amount of research has been dedicated to code cloning in recent years. The detailed surveys by Koschke [19] or Roy and Cordy [28] provide a comprehensive overview of existing work. Since this paper targets consequences of cloning and detection of inconsistent clones, we detail existing work in these areas.

3.1 Consequences of cloning

Indication for harmfulness of cloning for maintainability or correctness is given by several researchers. Lague et al. [23], report inconsistent evolution of a substantial amount of clones in an industrial telecommunication system. Monden et al. [27] report a higher revision number for files with clones than for files without in a 20 year old legacy system, possibly indicating lower maintainability. In [17], Kim et al. report that a substantial amount of changes to code clones occur in a coupled fashion, indicating additional maintenance effort due to multiple change locations.

Li et al. [25] present an approach to detect bugs based on inconsistent renaming of identifiers between clones. Jiang, Su and Chiu [12] analyze different contexts of clones, such as missing *if* statements. Both papers report the successful discovery of bugs in released software. In [1] and [2], individual cases of bugs or inconsistent bug fixes discovered by analysis of clone evolution are reported for open source software.

In contrast, doubt that consequences of cloning are unambiguously harmful is raised by several recent research results. Krinke [22] reports that only half the clones in several open source systems evolved consistently and that only

a small fraction of inconsistent clones becomes consistent again through later changes, potentially indicating a larger degree of independence of clones than hitherto believed. Geiger et al. [9] report that a relation between change couplings and code clones could, contrary to expectations, not be statistically verified. Lozano and Wermelinger [26] report that no systematic relationship between code cloning and changeability could be established.

The effect of cloning on maintainability and correctness is thus not clear. Furthermore, the above listed publications suffer from one or more shortcomings that limit the transferability of the reported findings.

- Instead of manual inspection of the actual inconsistent clones to evaluate consequences for maintenance and correctness, indirect measures[1] are used [1, 9, 22, 23, 26, 27]. Such approaches are inherently inaccurate and can easily lead to misleading results. For example, unintentional differences and faults, while unknown to developers, exhibit the same evolution pattern as intentional independent evolution and are thus prone to misclassification.
- The analyzed systems are too small to be representative [17] or omit analysis of industrial software [1, 2, 9, 17, 22, 26].
- The analyses specifically focus on faults introduced during creation [12, 25] or evolution [2] of clones, inhibiting quantification of inconsistencies in general.

Additional empirical research outside these limitations is required to better understand consequences of cloning [19, 28], as presented in this paper: Developer rating of the actual inconsistent clones has been performed, the study objects are both open source and industrial systems and inconsistencies have been analyzed independently of their mode of creation.

3.2 Detection of inconsistent clones

We classify existing approaches according to the program representation on which they operate.

Text Normalized code fragments are compared textually in a pairwise fashion [29]. A similarity threshold governs whether text fragments are considered as clones.

Token Ueda et al. [31] propose post-processing of the results of a token-based detection of exact clones. Essentially, neighboring exact clones are composed into inconsistent clones. In [25], Li et al. present the tool CP-Miner, which searches for similar basic blocks using frequent subsequence mining and then combines basic block clones into larger clones.

[1] Examples are change coupling or the ratio between consistent and inconsistent evolution of clones

Abstract Syntax Tree Baxter et al. [3] hash subtrees into buckets and perform pairwise comparison of subtrees in the same bucket. Jiang et al. [11] propose the generation of characteristic vectors for subtrees. Instead of pairwise comparison, they employ locality sensitive hashing for vector clustering, allowing for better scalability than [3]. In [7], tree patterns that provide structural abstraction of subtrees are generated to identify cloned code.

Program Dependence Graph Krinke [21] proposes a search algorithm for similar subgraph identification. Komondoor and Horwitz [18] propose slicing to identify isomorphic PDG subgraphs. Gabel, Jiang and Su [8] use a modified slicing approach to reduce the graph isomorphism problem to tree similarity.

The existing approaches provided valuable inspiration for the algorithm presented in this paper. However, none of them was applicable to our case study, for one or more of the following reasons.

- Tree [3,7,11] and graph [8,18,21] based approaches require the availability of suitable context free grammars for AST or PDG construction. While feasible for modern languages such as Java, this poses a severe problem for legacy languages such as Cobol or PL/I, where suitable grammars are not available. Parsing such languages still represents a significant challenge [5, 24].
- Due to the information loss incurred by the reduction of variable size code fragments to finite-size numbers or vectors, the edit distance between inconsistent clones cannot be precisely controlled in feature vector [11] and hashing based [3] approaches.
- Idiosyncrasies of some approaches threaten recall. In [31], inconsistent clones cannot be detected if their constituent exact clones are not long enough. In [8], inconsistencies might not be detected if they add data or control dependencies, as noted by the authors.
- Scalability to industrial-size software of some approaches has been shown to be infeasible [18, 21] or is at least still unclear [7, 29].
- For most approaches, implementations are not publicly available.

In contrast, the approach presented in this paper supports both modern and legacy languages including Cobol and PL/I, allows for precise control of similarity in terms of edit distance on program statements, is sufficiently scalable to analyze industrial-size projects in reasonable time and is available for use by others as open source software.

An approach similar to [31] for bug detection has been outlined by the authors of this paper in [15]. In contrast to this work, it does not use a suffix tree based algorithm and no empirical study was performed.

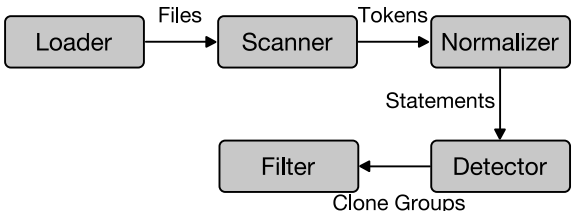

Figure 2. The clone detection pipeline used

4. Detecting inconsistent clones

This section explains the approach used for detecting inconsistent clones in large amounts of code. Our approach works on the token level, which usually is sufficient for finding copy-pasted code, while at the same time being efficient. The algorithm works by constructing a suffix tree of the code and then for each possible suffix an approximate search based on the edit distance in this tree is performed.

Our clone detector is organized as a pipeline, which is sketched in Figure 2. The files under analysis are loaded and then fragmented by the *scanner*, yielding a stream of tokens, which is filtered to exclude comments and generated code (recognized by user provided patterns). From the token stream, which consist of single keywords, identifiers, operators, and so on, the *normalizer* reassembles statements. This stage performs normalization, such that differences in identifier names or constant values are not relevant when comparing statements. The sequence formed by those statements is then fed into our clone detection algorithm, which finds and reports clone groups in this stream. Finally, clone groups are post-processed and uninteresting ones are filtered out. We outline the detection steps in more detail in the following subsections.

4.1. Preprocessing and normalization

As stated before, the code is read and split into tokens using a scanner. An important task during preprocessing is normalization, which creates statements from the scanner's tokens. This is used as it allows better tailoring of normalization and to avoid clones starting or ending within statements. The used normalization eliminates differences in naming of identifiers and values of constants or literals, but does not, for example, change operation order.

Further tasks of the preprocessing phase are the removal of comments or generated code, which is either already excluded at the file level or on the token stream based on certain patterns that recognize sections of generated code.

4.2. Detection algorithm

The task of the detection algorithm is to find clones in the stream of units provided by the normalizer. Stated differently, we want to find common substrings in the sequence formed by all units of the stream, where common substrings are not required to be exactly identical (after normalization), but may have an edit distance bounded by some threshold. This problem is related to the approximate string matching problem [13, 32], which is also investigated extensively in bioinformatics [30]. The main difference is that we are not interested in finding an approximation of only a single given word in the string, but rather are looking for *all* substrings approximately occurring more than once in the entire sequence.

A sketch of our detection algorithm is shown in Figs. 3 and 4. The algorithm is an edit distance based traversal of a suffix tree of our input sequence. A suffix tree over a sequence s is a tree with edges labeled by words such that exactly all suffixes of s are found by traversing the tree from the root node to a leaf and concatenating the words on the edges encountered. Such a suffix tree can be constructed in linear time by the well-known online algorithm by Ukkonen [33]. Using this suffix tree, we start a search for clones at every possible index.

Searching for clones is performed by the procedure *search* which recursively traverses the suffix tree. The first two parameters to this function are the sequence s we are working on and the position *start* where the search was started, which is required when reporting a clone. The parameter j (which is the same as *start* in the first call of *search*) marks the current end of the substring under inspection. To prolong this substring, the substring starting at j is compared to the word w being next in the suffix tree, which is the edge leading *to* the current node v (for the root node we just use the empty string). For this comparison an edit distance of at most e operations (fifth parameter) is allowed. For the first call of *search*, e is the edit distance maximally allowed for a clone. If the remaining edit operations are not enough to *match* the entire edge word w (else case), we report the clone as far as we found it, otherwise the traversal of the tree continues recursively, increasing the length ($j-start$) of the current substring and reducing the number e of edit operations available by the amount of operations already spent in this step.

To actually make this algorithm work and its results usable, some details have to be fleshed out. For the computation of the longest edit distance match we are using the simple dynamic programming algorithm found in algorithm textbooks. While this is easy to implement, it requires quadratic time and space[2]. To make this step work

[2]Actually the algorithm can be implemented using only linear space, but preserving the full calculation matrix allows us some simplifications.

```
proc detect (s, e)
Input: String s = (s_0, ..., s_n), max edit distance e
1  Construct suffix tree T from s
2  for each i ∈ {1, ..., n} do
3     search (s, i, i, root(T), e)
```

Figure 3. Outline of approximate clone detection algorithm

```
proc search (s, start, j, v, e)
Input: String s = (s_0, ..., s_n),
   start index of current search, current search index j,
   node v of suffix tree over s, max edit distance e
1  Let (w_1, ..., w_m) be the word along the edge leading to v
2  Calculate the maximal length l ≤ m, such that
   there is a k ≥ j where the edit distance e' between
   (w_1, ..., w_l) and (s_j, ..., s_k) is at most e
3  if l = m then
4     for each child node u of v do
5        search (s, start, k + m, u, e − e')
6  else if k − start ≥ minimal clone length then
7     report substring from start to k of s as clone
```

Figure 4. Search routine of the approximate clone detection algorithm

efficiently we look at most at the first 1000 statements of the word w. As long as the word on the suffix tree edge is shorter, this is not a problem. In case there is a clone of more than 1000 statements, we would find it in chunks of 1000. We considered this to be tolerable for practical purposes. As each suffix we are running the search on will of course be part of the tree, we also have to make sure that no self matches are reported.

When running the algorithm as it is, the results are often not as expected because the search tries to match as many statements as possible. However, allowing for edit operations right at the beginning or at the end of a clone is not helpful, as then every exact clone can be prolonged into an inconsistent clone. Thus in the search we enforce the first few statements (how many is parameterized) to match exactly. (This also speeds up the search, as we can choose the correct child node at the root of the suffix tree in one step without looking at all children.) The last statements are also not allowed to differ, which is checked for and corrected just before reporting a clone.

Including all of these optimizations, the algorithm can miss a clone either due to the thresholds (either too short or too many inconsistencies), or if it is covered by other clones. The later case is important, as each substring of a clone of course is a clone again and we usually do not want these to be reported.

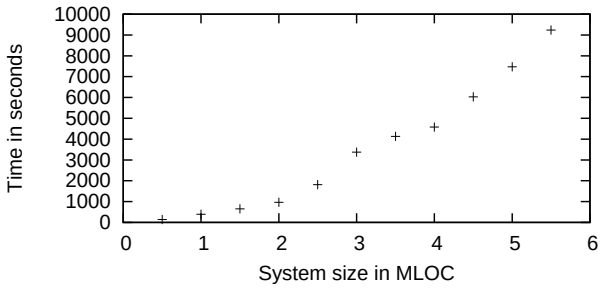

Figure 5. Runtime of inconsistent clone detection on Eclipse source

4.3. Post-processing and filtering

During and after detection, the clone groups that are reported are subject to filtering. Filtering is usually performed as early as possible, so no memory is wasted with storing clone groups that are not considered relevant. Using these filters, we discard clone groups whose clones overlap with each other and groups whose clones are contained in other clone groups. Additionally, we enforce not only an absolute limit on the number of inconsistencies, but also a relative one, *i. e.*, we filter clone groups where the number of inconsistencies in the clones relative to the clone's length exceeds a certain amount. Moreover, we merge clone groups which share a common clone. While this leads to clone groups with non related clones (as our definition of an inconsistent clone is not transitive), for practical purposes it is preferred to know of these indirect relationships, too.

4.4. Tool support

To be able to experiment with the detection of inconsistent clones, our algorithms and filters have been implemented as part of CloneDetective[3] [14] which is based on ConQAT [**?**]. The result is a highly configurable and extensible platform for clone detection on the syntactic level. As our cloning pipeline could reuse a major portion of the CloneDetective code, we consider such an open platform essential for future experiments, as it allows researchers to focus on individual parts of the pipeline. CloneDetective also offers a front-end to visualize and assess the clones found, and thus supports the rapid review of a large number of clone groups.

4.5. Scalability and performance

Due to the many implementation details, the worst case complexity is hard to analyze. Additionally, for practical

[3]Available as Open Source at `http://conqat.cs.tum.edu/CloneDetective`

purposes, the more complicated average complexity would be more adequate. Thus, and to assess the performance of the entire pipeline we executed the detector on the source code of Eclipse[4], limiting detection to a certain amount of code. Our results on an Intel Core 2 Duo 2.4 GHz running Java in a single thread with 3.5 GB of RAM are shown in Figure 5. The settings are the same as for the main study (min clone length of 10, max edit distance of 5). It is capable to handle the 5.6 MLOC of Eclipse in about 3 hours, which is fast enough to be executed within a nightly build.

5. Study description

In order to gain a solid insight into the effects of inconsistent clones, we use a study design with 5 objects and 3 research questions that guide the investigation.

5.1. Study objects

We chose 2 companies and 1 open source project as sources of software systems. This resulted in 5 analyzed projects in total. We chose systems written in different languages, by different teams in different companies and with different functionalities to increase the transferability of the study results. These objects included 3 systems written in C#, a Java system as well as a long-lived Cobol system. All these systems are already in production. For non-disclosure reasons we gave the commercial systems names from A to D. An overview is shown in Table 1.

Munich Re Group The Munich Re Group is one of the largest re-insurance companies in the world and employs more than 37,000 people in over 50 locations. For their insurance business, they develop a variety of individual supporting software systems. In our study, we analyzed the systems A, B and C, all written in C#. They were each developed by different organizations and provide substantially different functionality, ranging from damage prediction, over pharmaceutical risk management to credit and company structure administration. The systems support between 10 and 150 expert users each.

LV 1871 The Lebensversicherung von 1871 a.G. (LV 1871) is a Munich-based life-insurance company. The LV 1871 develops and maintains several custom software systems for mainframes and PCs. In this study, we analyze a mainframe-based contract management system written in Cobol (System D) employed by about 150 users.

[4]Core of Eclipse Europa release 3.3

Sysiphus The open source system *Sysiphus*[5] is developed at the Technische Universität München (TUM) but none of the authors of this paper have been involved in the development. It constitutes a collaboration environment for distributed software development projects. The inclusion of an open source system is motivated by the fact that, as the clone detection tool is also freely available, the results can be externally replicated[6]. This is not possible with the detailed confidential results of the commercial systems.

Table 1. Summary of the analyzed systems

System	Organization	Language	Age (years)	Size (kLOC)
A	Munich Re	C#	6	317
B	Munich Re	C#	4	454
C	Munich Re	C#	2	495
D	LV 1871	Cobol	17	197
Sysiphus	TUM	Java	8	281

5.2. Research questions

The underlying problem that we analyze are clones and especially their inconsistencies. In order to investigate this question, we answer the following 3 more detailed research questions.

RQ 1 *Are clones changed inconsistently?*

The first question we need to answer is whether inconsistent clones appear at all in real-world systems. This not only means whether we can find them at all but also whether they constitute a significant part of the total clones of a system. It does not make sense to analyze inconsistent clones if they are a rare phenomenon.

RQ 2 *Are inconsistent clones created unintentionally?*

Having established that there are inconsistent clones in real systems, we need to analyze whether these inconsistent clones have been created intentionally or not. It can obviously be sensible to change a clone so that it becomes inconsistent to it counterparts because it has to conform to different requirements. However, the important difference is whether the developer is aware of the other clones, i.e. whether the inconsistency is intentional.

RQ 3 *Can inconsistent clones be indicators for faults in real systems?*

[5]`http://sysiphus.in.tum.de/`
[6]`http://wwwbroy.in.tum.de/~ccsm/icse09/`

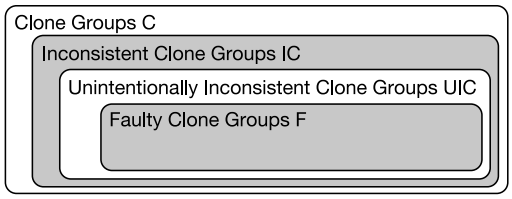

Figure 6. Clone Group Sets

After establishing these prerequisites, we can determine whether the inconsistent clones are actually indicators for faults in real systems. If there are inconsistent clones that have not been created because of different requirements, this implies that at least one of these clones does not conform to the requirements. Hence, it constitutes a fault.

5.3. Study design

We answer the research questions with the following study design. In the study we analyze sets of clone groups as shown in Fig. 6. The outermost set are all clone groups C in a system, *IC* denotes the set of inconsistent clone groups, and *UIC* the unintentionally inconsistent clone groups. The subset F of *UIC* consists of those unintentionally inconsistent clone groups that indicate a fault in the program. Please note that we do not distinguish between *created* and *evolved* inconsistent clones as for the question of faultiness it does not matter when the inconsistencies have been introduced.

We use these different clone group sets to design the study that answers our research questions. The independent variables in the study are development team, programming language, functional domain, age and size. The dependent variables for the research questions are explained below. RQ 1 investigates the existence of inconsistent clones in realistic systems. Hence, we need to analyze the size of set *IC* with respect to the size of set C. We apply our inconsistent clone analysis approach to all the systems, perform manual assessment of the detected clones to eliminate false positives and calculate the *inconsistent clone ratio* $|IC|/|C|$.

For RQ 2, whether clones are created unintentionally, we then compare the size of the sets *UIC* and *IC*. The sets are established by showing each identified inconsistent clone to developers of the system and asking them to rate them as intentional or unintentional. This gives us the *unintentionally inconsistent clone ratio* $|UIC|/|IC|$. The most important question we aim to answer is whether inconsistent clones indicate faults (RQ 3). Hence, we are interested in the size of set F in relation to the size of *IC*. The set F is again determined by asking developers of the respective system. Their expert opinion classifies the clones in faulty and non-faulty. We only analyze unintentionally inconsistent clones for faults. Our *faulty inconsistent clone ratio* $|F|/|IC|$ is thus a lower bound, as potential faults in intentionally inconsistent clones are not considered.

Using this, we are already able to roughly find the answer to RQ 3. As this is our main result from the study, we transform it into a hypothesis. We need to make sure that the fault density in the inconsistencies is higher than in randomly picked lines of source code. This leads to the hypothesis H:

The fault density in the inconsistencies is higher than the average fault density.

As we do not know the actual fault densities of the analyzed systems, we need to resort to average values. The span of available numbers is large because of the high variation in software systems. Endres and Rombach [6] give 0.1–50 faults per kLOC as a typical range. For the fault density in the inconsistencies, we use the number of faults divided by the logical lines of code of the inconsistencies. We refrain from testing the hypothesis statistically because of the low number of data points as well as the large range of typical defect densities.

5.4. Procedure

The treatment we used on the objects was the approach to detect inconsistent clones as described in section 4. For all systems, the detection was executed by the researcher to identify consistent and inconsistent clone candidates. On an 1.7 GHz notebook, the detection took between one and two minutes for each system. The detection was configured to not cross method boundaries, since experiments showed that inconsistent clones that cross method boundaries in many cases did not capture semantically meaningful concepts. This is also noted for exact clones in [20] and is even more pronounced for inconsistent clones. Since in Cobol sections in the procedural division are the counterpart of Java or C# methods, clone detection for Cobol was limited to these.

For the C# and Java systems, the algorithm was parameterized to use 10 statements as minimal clone length, a maximum edit distance of 5, a maximal inconsistency ratio (*i. e.*, the ratio of edit distance and clone length) of 0.2 and the constraint that the first 2 statements of two clones need to be equal. Due to the verbosity of Cobol [5], minimal clone length and maximal edit distance were doubled to 20 and 10, respectively. Generated code that is not subject to manual editing was excluded from clone detection, since inconsistent manual updates obviously cannot occur. Normalization of identifiers and constants was tailored as appropriate for the analyzed language, to allow for renaming of identifiers while at the same time avoiding too large false positive rates. These settings were determined to represent the best compromise between precision and recall during cursory experiments on the analyzed systems, for which

Table 2. Summary of the study results

Project	A	B	C	D	Sysiphus	Sum	Mean
Precision exact clone groups	0.88	1.00	0.96	1.00	0.98	—	0.96
Precision inconsistent clone groups	0.61	0.86	0.80	1.00	0.87	—	0.83
Clone groups $\|C\|$	286	160	326	352	303	1427	—
Inconsistent clone groups $\|IC\|$	159	89	179	151	146	724	—
Unintentionally inconsistent clone groups $\|UIC\|$	51	29	66	15	42	203	—
Faulty clone groups $\|F\|$	19	18	42	5	23	107	—
RQ 1 $\|IC\|/\|C\|$	0.56	0.56	0.55	0.43	0.48	—	0.52
RQ 2 $\|UIC\|/\|IC\|$	0.32	0.33	0.37	0.10	0.29	—	0.28
RQ 3 $\|F\|/\|IC\|$	0.12	0.20	0.23	0.03	0.16	—	0.15
Faulty in UIC $\|F\|/\|UIC\|$	0.37	0.62	0.64	0.33	0.55	—	0.50
Inconsistent logical lines	442	197	797	1476	459	3371	—
Fault density in kLOC^{-1}	43	91.4	52.7	3.4	50.1	—	48.1

random samples of the detected clones have been evaluated manually.

The detected clone candidates were then manually rated by the researcher in order to remove false positives, *i. e.*, code fragments that, although identified as clone candidates by the detection algorithm, have no semantic relationship. Inconsistent and exact clone group candidates were treated differently: *all* inconsistent clone group candidates were rated, producing the set of inconsistent clone groups. Since the exact clones were not required for further steps of the case study, instead of rating all of them, a random sample of 25% was rated, and false positive rates then extrapolated to determine the number of exact clones.

The inconsistent clone groups were then presented to the developers of the respective systems in the tool *CloneDetective* mentioned in Section 4.4, which is able to display the commonalities and differences of the clone group in a clearly arranged way, as depicted in Figs. 1 and 7. The developers rated whether the clone groups were created intentionally or unintentionally. If a clone group was created unintentionally, the developers also classified it as faulty or non-faulty. For the Java and C# systems, all inconsistent clone groups were rated by the developers. For the Cobol system, rating was limited to a random sample of 68 out of the 151 inconsistent clone groups, since the age of the system and the fact that the original developers were not available for rating increased rating effort. Thus, for the Cobol case, the results for RQ 2 and RQ 3 were computed based on this sample. In cases where intentionality or faultiness could not be determined, *e. g.*, because none of the original developers could be accessed for rating, the inconsistencies were treated as intentional and non-faulty.

6. Results

The quantitative results of our study are summarized in Table 2. Except for the Cobol system D, the precision values are smaller for inconsistent clone groups than for exact clone groups, as was expected, since inconsistent clone groups allow for more deviation. The high precision results of system D result from the rather conservative clone detection parameters chosen due to the verbosity of Cobol. For system A, stereotype database access code of semantically unrelated objects gave rise to lower precision values.

About half of the clones (52%) contain inconsistencies. Therefore, RQ 1 can be positively answered: Clones are changed inconsistently. All these would not be reported by existing tools that search for exact matches. From these inconsistencies over a quarter (28%) has been introduced unintentionally. Hence, RQ 2 can also be answered positively: Inconsistent clones are created unintentionally in many cases. Only system D is far lower here, with only 10% of unintentionally inconsistent clones. With about three quarters of intentional changes, this shows that cloning and changing code seems to be a frequent pattern during development and maintenance.

For RQ 3, whether inconsistent clones are indicators for faults, we note that at least 3-23% of the inconsistencies actually presented a fault. Again the by far lowest number comes from the Cobol system. Ignoring it, the total ratio of faulty inconsistent clones goes up to 18%. This constitutes a significant share that needs consideration. To judge hypothesis H, we also calculated the fault densities. They lie in the range of 3.4–91.4 faults per kLOC. Again, system D is an outlier. Compared to reported fault densities in the range of 0.1 to 50 faults and considering the fact that all systems are not only delivered but even have been productive for several years we consider our results to support hypothesis H. On average the inconsistencies contain more faults than average code. Hence, RQ 3 can also be answered positively: Inconsistent clones can be indicators for faults in real systems.

While the numbers are similar for the C# and Java projects, rates of unintentional inconsistencies and thus

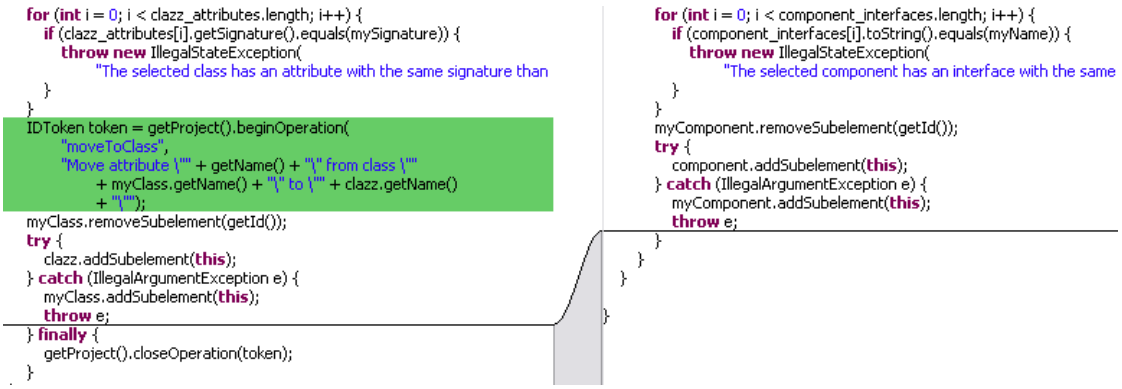

Figure 7. Different UI behavior since right side does not use operations (Sysiphus).

faults are comparatively low for project D, which is a legacy system written in Cobol. To a certain degree, we attribute this to our conservative assessment strategy of treating inconsistencies whose intentionality and faultiness could not be unambiguously determined as intentional and non-faulty. Furthermore, interviewing the current maintainers of the systems revealed that cloning is such a common pattern in Cobol systems, that searching for duplicates of a piece of code is actually an integral part of their maintenance process. Compared to the developers of the other projects, the Cobol developers where thus more aware of clones in the system. To account for this difference in "clone awareness" we added the row $|F|/|UIC|$ to Table 2, which reveals that while the rates of unintentional changes are lower for project D, the ratio of unintentional changes leading to a fault is in the same range for all projects. From our results it seems that about every second to third unintentional change to a clone leads to a fault.

Although not central to our research questions, the detection of faults almost automatically raises the question for their severity. As the fault effect costs are unknown for the analyzed systems, we cannot provide a full-fledged severity classification. However, we provide a partial answer by categorizing the found faults as (1) faults that lead to potential system crash or data loss, (2) unexpected behavior visible to the end user and (3) unexpected behavior not visible to the end user. One example for a category (1) fault is shown in Fig 1. Here, one clone of the affected clone group performs a null-check to prevent a null-pointer dereference whereas the other does not. Other examples we encountered for category (1) faults are index-out-of-bounds exceptions, incorrect transaction handling and missing rollbacks. Fig. 7 shows an example of a category (2) fault. In one clone the performed operation is not encapsulated in an operation object and, hence, is handled differently by the undo mechanism. Further examples we found for category (2) faults are incorrect end user messages, inconsistent default values as well as different editing and validation behavior in similar user forms and dialogs. Category (3) examples we identified include unnecessary object creation, minor memory leaks, performance issues like missing break statements in loops and redundant re-computations of cache-able values, differences in exception handling, different exception and debug messages or different log levels for similar cases. Of the 107 inconsistent clones found, 17 were categorized as category (1) faults, 44 as category (2) faults and 46 as category (3) faults. Since all analyzed systems are in production, the relatively larger amounts of category (2) and (3) faults coincide with our expectations.

7. Threats to validity

We discuss how we mitigated threats to construct, internal and external validity of our study.

7.1. Construct validity

We did not analyze the development repositories of the systems in order to determine if the inconsistencies really have been introduced by incomplete changes to the system and not by random similarities of unrelated code. This has two reasons: (1) We want to analyze all inconsistent clones, also the ones that have been introduced directly by copy and modification in a single commit. Those might not be visible in the repository. (2) The industrial systems do not have complete development histories. We confronted this threat by manually analyzing each potential inconsistent clone.

The comparison with average fault probability is not perfect to determine whether the inconsistencies are really more fault-prone than a random piece of code. A comparison with the actual fault densities of the systems or actual checks for faults in random code lines would better suit this purpose. However, the actual fault densities are not available to us because of incomplete defect databases. To check for faults in random code lines is practically not possible.

We would need the developers time and willingness for inspecting random code. As the potential benefit for the developers is low, the motivation would be low and hence the results would be unreliable.

7.2. Internal validity

As we ask the developers for their expert opinion on whether an inconsistency is intentional or unintentional and faulty or non-faulty, a threat is that the developers do not judge this correctly. One case is that the developer assesses something as non-faulty which actually is faulty. This case only reduces the chances to positively answer the research questions. The second case is that the developers rate something as faulty which is no fault. We mitigated this threat by only rating an inconsistency as faulty if the developer was completely sure. Otherwise it was postponed and the developer consulted colleagues that know the corresponding part of the code better. Inconclusive candidates were ranked as intentional and non-faulty. Hence, again only the chance to answer the research question positively is reduced.

The configuration of the clone detection tool has a strong influence on the detection results. We calibrated the parameters based on a pre-study and our experience with clone detection in general. The configuration also varies over the different programming languages encountered, due to their differences in features and language constructs. However, this should not strongly affect the detection of inconsistent clones because we spent great care to configure the tool in a way that the resulting clones are sensible.

We also pre-processed the inconsistent clones that we presented to the developers in order to eliminate false positives. This could mean that we excluded clones that are actually faulty. However, this again only reduces the chances that we can answer our research question positively.

7.3. External validity

The projects were obviously not sampled randomly from all possible software systems but we relied on our connections with the developers of the systems. Hence, the set of systems is not completely representative. The majority of the systems is written in C# and analyzing 5 systems in total is not a high number. However, all 5 systems have been developed by different development organizations and the C#-systems are technically different (2 web, 1 rich client) and provide substantially different functionalities. We further mitigated this threat by also analyzing a legacy Cobol system as well as an open source Java system.

8. Discussion

Even considering the threats to validity discussed above, the results of the study show convincingly that clones can lead to faults in a system. The inconsistencies between clones are often not justified by different requirements but can be explained by developer mistakes.

We consider of special value the analysis of the Sysiphus project. Because both Sysiphus and our detection tools are open source, the whole analysis can completely be replicated independently. We provide a web site with the necessary information[7].

Having established the empirical results, the question remains of how to use this information in order to reduce faults in software systems. The answer is twofold: (1) prevention by less cloning and (2) tools that prevent unintentionally inconsistent changes of clones. The fewer clones there are in the system, the less likely it is to introduce faults by inconsistencies between them. In order to increase developer awareness of clones, we have integrated our clone detection tool into the Visual Studio development environment[8]. At the Munich Re Group, as a reaction on the clone results, clone detection is now included in the nightly builds of all discussed projects. Furthermore, for existing clones, there should be tool support that ensures that all changes that are made to a clone are made in the full knowledge of its duplicates. Tools such as CloneTracker [4] or CReN [10] provide promising approaches. However, both approaches are not applicable to existing software that already contains inconsistent clones. Due to their high fault potential, we consider the ability to detect inconsistent clones an important feature of industrial-strength clone detectors.

9. Conclusion

In this paper we provide strong evidence that inconsistent clones constitute a major source of faults, which means that cloning can be a substantial problem during development and maintenance unless special care is taken to find and track existing clones and their evolution. Our results suggest that nearly every second unintentionally inconsistent change to a clone leads to a fault. Furthermore, we provide a scalable algorithm for finding such inconsistent clones as well as suitable tool support for future experiments.

Future work on this topic will evolve in multiple directions. One obvious development is the refinement of the algorithms and tools used. This includes refined heuristics to speed up the clone search and perform automatic assessment to discard obviously irrelevant clones. In addition, the usability of the tools could be advanced further to make their use more efficient for practical applications. Moreover, it will be interesting to compare different detection parameter values, algorithms and tools according to their performance and accuracy when finding inconsistent clones.

[7] http://wwwbroy.in.tum.de/~ccsm/icse09/

[8] http://www.codeplex.com/CloneDetectiveVS

Additionally, while answering some questions, our data of course raises a couple of new relevant questions. One is a more detailed quantitative classification of defect types of the faults found. Another question is whether those faults are also detected by classical techniques such as dynamic testing. However, to answer these questions the developers of the analyzed systems have to be interviewed again.

The underlying major question is how studying cloning can help in reducing the development and maintenance costs of software systems. This paper takes a first step into this direction, but more work needs to be done to develop a usable and economically sensible methodology.

Coming back to the paper title, we found that code clones do matter. Our result is, however, limited to the consequences of clones on program correctness. Hence, we believe that the most important task of future work is to investigate the impact of clones on software maintenance effort.

Acknowledgments The authors would like to thank the Munich Re Group, LV 1871 and the Sysiphus team for supporting this study as well as Magne Jørgensen for helpful comments on the empirical analysis. This work has partially been supported by the German Federal Ministry of Education and Research (BMBF) in the project QuaMoCo (01 IS 08023B).

References

[1] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proc. CSMR '07*. IEEE, 2007.

[2] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *Proc. ICSM '07*. IEEE, 2007.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. ICSM '98*. IEEE, 1998.

[4] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. ICSE '07*. IEEE, 2007.

[5] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. ICSM '99*. IEEE, 1999.

[6] A. Endres and D. Rombach. *A Handbook of Software and Systems Engineering*. Pearson, 2003.

[7] W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. In *Proc. WCRE '07*. IEEE, 2007.

[8] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proc. ICSE '08*. ACM, 2008.

[9] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proc. FASE06*. Springer, 2006.

[10] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proc. Eclipse '07*. ACM, 2007.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Decard: Scalable and accurate tree-based detection of code clones. In *Proc. ICSE '07*. IEEE, 2007.

[12] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proc. ESEC-FSE '07*. ACM, 2007.

[13] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. MFCS '91*, volume 520 of *LNCS*. Springer, 1991.

[14] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective: A workbench for clone detection research. In *In proc. of ICSE 2009*, 2009.

[15] E. Juergens, B. Hummel, F. Deissenboeck, and M. Feilkas. Static bug detection through analysis of inconsistent clones. In *Workshopband SE Konferenz 2008*, LNI. GI, 2008.

[16] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *Proc. WCRE '06*. IEEE, 2006.

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. ESEC/FSE-13*. ACM, 2005.

[18] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. SAS '01*, volume 2126 of *LNCS*. Springer, 2001.

[19] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.

[20] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. WCRE '06*. IEEE, 2006.

[21] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. WCRE '01*. IEEE, 2001.

[22] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. WCRE '07*. IEEE, 2007.

[23] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. ICSM '97*. IEEE, 1997.

[24] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Softw. Pract. Exp.*, 31(15):1395–1438, 2001.

[25] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng*, 32(3):176–192, 2006.

[26] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *ICSM 2008*. IEEE, 2008.

[27] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proc. METRICS '02*. IEEE, 2002.

[28] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen's University at Kingston, 2007.

[29] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. ICPC '08*. IEEE, 2008.

[30] H. Täubig. *Fast Structure Searching for Computational Proteomics*. PhD thesis, TU München, 2007.

[31] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *Proc. APSEC '02*, 2002.

[32] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM '93*, volume 684 of *LNCS*. Springer, 1993.

[33] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.