ABSTRACT

Code Clones—duplicated source fragments—are said to increase maintenance effort and to facilitate problems caused by inconsistent changes to identical parts. While this is certainly true for some clones and certainly not true for others, it is unclear how many clones are real threats to the system’s quality and need to be taken care of. Our analysis of clone evolution in mature software projects shows that most clones are rarely changed and the number of unintentional inconsistent changes to clones is small. We thus have to carefully select the clones to be managed to avoid unnecessary effort managing clones with no risk potential.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—restructuring, reverse engineering, and reengineering

General Terms
Experimentation, Measurement

Keywords
Software maintenance, clone detection, clone evolution

1. INTRODUCTION

Code clones are similar fragments of source code. There are many problems caused by the presences of clones. Among others, the source code becomes larger, change effort increases, and change propagation bears the risk of unwanted inconsistencies—for example, incomplete removal of defects. Consequently, a variety of clone detection techniques and tools has evolved to identify duplicated source code within a system. In addition, various tools have been created that support developers in managing clones. These include refactoring support [4, 12], automated change propagation [6, 33], and change monitoring to prevent unintentional inconsistencies [30].

There certainly exist clones that are true threats to software maintenance. Nevertheless, recent research [19, 20] doubts the harmfulness of clones in general and lists numerous situations in which clones are a reasonable design decision. From the clone management perspective, it is desirable to detect and manage only the harmful clones, because managing clones that have no negative effects creates only additional effort.

Unfortunately, state-of-the-art clone tools detect and classify clones based only on similar structures in the source code or one of its various representations. When it comes to clone-related problems, however, the most important characteristic of a clone is its change behavior and not its structure. Only if a clone changes, it causes additional change effort. Only if a clone changes, unintentional inconsistencies can arise. If, on the other hand, a clone never changes, there are no additional costs induced by propagating changes and there is no risk of unwanted inconsistencies.

Our hypothesis is that many clones detected by state-of-the-art tools are “structurally interesting” but irrelevant to software maintenance because they never change during their lifetime.

Up-to-date clone detectors can efficiently process and detect clones within huge amounts of source code, consequently delivering huge numbers of clones. In contrast, clone assessment and deciding how to proceed can be very costly even for individual clones as we have experienced with clones in our own code [11]. Hence, having many unproblematic clones in the detection results creates enormous overhead for assessing and managing clones that do not threaten maintenance because they never change.

To gain a better understanding of clones’ threat potential, we conducted an extensive study on clone evolution in different systems and performed a detailed tracking to detect when and how clones had been changed. For this study, we concentrated on two prominent clone-related problems—the additional change effort caused by clones and the risk of unintentional inconsistent changes. Our research questions are the following:

Question 1 — How often are clones changed throughout their lifetime?

Question 2 — How many changes to clones are unintentionally inconsistent?

Contribution. The contribution of our work is a detailed analysis of how individual clones were changed throughout their lifetime.
the risk of unintentional inconsistencies. We provide an evaluation of the threat potential of clones detected by a state-of-the-art tool.

Outline. The structure of this paper is as follows. Section 2 introduces important terminology and concepts. Related work is given in Section 3. Section 4 describes our technique to extract the necessary clone evolution data. The setup of our study is outlined in Section 5. The data we collected and our results are presented in Section 6. We discuss our results in Section 7. Our conclusions are provided in Section 8.

2. TERMINOLOGY

In this section, we introduce relevant terminology. Clone detection, representation, and analysis can be done on different abstractions of a system, each of which has advantages and disadvantages. Approaches may be based on text [7, 15], tokens [9, 18], abstract syntax trees [5, 13], graphs [21, 23], or metrics [28]. We use a token-based approach that treats the system’s files as sequences of tokens. Token-based approaches are language independent—except for the lexer that translates source text into a sequence of tokens—and scalable while maintaining a basic understanding of the program’s structure.

A fragment is a contiguous section of source code. As such, a fragment has a well-defined start and end and is contained in exactly one file in exactly one version of the system. A fragment has an arbitrary number of ancestors which are its occurrences in the previous version of the system. Each fragment has zero or one descendant which is its occurrence in the next version of the system.

A clone pair relates two fragments that are similar to each other and are contained in the same version of the system. The degree of similarity is expressed by different types:

Type 1. The token sequences of both fragments are identical disregarding whitespace and comments.

Type 2. A type-1 clone pair, but values of identifiers and literals may differ.

Type 3. A type-2 clone pair with gaps, that is, some tokens exist in only one of the fragments.

In contrast to a clone pair, a clone class groups two or more similar fragments. Like a clone pair, we assign one of the three types to a clone class to describe the similarity between its fragments. Regarding type-3 clone classes, it is important that the clone-class relation is not transitive—between its fragments. Regarding type-3 clone classes, it is important that the clone-class relation is not transitive—between its fragments. The consistency of changes to clones has been quantified by Kim and her colleagues [20]. They analyzed the lifetime, occurrence of consistent changes, and suitability for refactoring of clone genealogies in two small Java systems. Although their study has been a major inspiration, our work differs in the way that we use a more detailed clone tracking (based on individual fragments) and we analyze changes for individual genealogies in more detail.

Aversano and her colleagues [1] extended Kim’s study with focus on the consistency of changes. They inspected individual changes and classified them according to whether clones evolved independently or were unknowingly changed inconsistently. In contrast to our study, they inspected individual changes not considering repeated changes to the same clones. Furthermore, clones that never changed were not considered.

Bakota and colleagues [3] analyzed change anomalies in the evolution of clones. They map individual fragments instead of clones between versions. In their case study, they found 60 anomalies in the evolution of Mozilla Firefox of which six have been classified as true defects. The change frequency of clones has not been investigated.

The consistency of changes to clones has been quantitatively investigated by Kriinke [24]. Analyzing changes to clones in five different systems, he found that the number of consistent changes and inconsistent changes are roughly the same. Clones that never change, repeated changes to class. A dark shade indicates an inconsistent change, that is, at least one fragment has been changed, but not all fragments have been changed in the same way. Fragments are connected to their ancestors and descendants by lines (normal edges).

A clone genealogy is a connected component in the clone evolution graph. As such, it is a set of classes all of which are connected by the ancestry of their fragments. For example, the graph in Figure 1 shows four clone genealogies.

3. RELATED WORK

This section presents previous research related to ours regarding the impact of clones on maintainability. For a general overview of research on software clones please refer to [22, 31]. Previous work can be roughly classified into two categories. First, studies targeted at clone evolution and the ways in which clones change. Second, studies analyzing the harmfulness of clones.

3.1 Clone Evolution

One of the first studies on clone evolution has been conducted by Kim and her colleagues [20]. They analyzed the lifetime, occurrence of consistent changes, and suitability for refactoring of clone genealogies in two small Java systems. Although their study has been a major inspiration, our work differs in the way that we use a more detailed clone tracking (based on individual fragments) and we analyze changes for individual genealogies in more detail.

Aversano and her colleagues [1] extended Kim’s study with focus on the consistency of changes. They inspected individual changes and classified them according to whether clones evolved independently or were unknowingly changed inconsistently. In contrast to our study, they inspected individual changes not considering repeated changes to the same clones. Furthermore, clones that never changed were not considered.

Bakota and colleagues [3] analyzed change anomalies in the evolution of clones. They map individual fragments instead of clones between versions. In their case study, they found 60 anomalies in the evolution of Mozilla Firefox of which six have been classified as true defects. The change frequency of clones has not been investigated.

The consistency of changes to clones has been quantitatively investigated by Kriinke [24]. Analyzing changes to clones in five different systems, he found that the number of consistent changes and inconsistent changes are roughly the same. Clones that never change, repeated changes to
the same clones, and harmfulness of changes have not been analyzed.

In one of our previous studies [8] we analyzed clone evolution in nine different systems. Although we performed a quantitative comparison of change consistency, we did neither analyze the frequency of changes for individual clones nor evaluate the harmfulness of changes.

Thummalapenta and colleagues [32] analyzed change patterns in the evolution of code clones. They found that most clones were consistently changed or deliberately evolved independently. Hence, most inconsistent changes were intentional. The change frequency of clones has not been considered.

While changes to clones have been analyzed in various studies, the clones that never change have been neglected. Furthermore, repeated changes of the same clone have not been analyzed so far.

3.2 Harmfulness of Clones

While it is frequently stated that clones negatively affect software maintenance, there is not much proof that they really do. Regarding the problem of clones increasing the effort to change the code, there is hardly any work to quantify the additional effort. While it is not known how much additional change effort is caused by clones and how many clones are responsible for this, previous studies [1, 8, 20, 24, 32] have shown that some additional effort must exists, since numerous consistent changes to clones have been detected.

Lozano and her colleagues [27] analyzed whether functions change more often when having clones. Their initial results suggest that functions change more often when they have clones compared to when they do not have clones. Nevertheless, the following study by Lozano and Wermeling [26] showed that change effort did not increase for more than half of the methods despite having clones. Only recently, Juergens and Deissenboeck [16] presented an analytical model to quantify the additional maintenance costs caused by clones. The model has not been evaluated yet.

Regarding the risk of unintentional inconsistent changes to clones, different studies have shown that this problem exists. Li and colleagues [25] identified defects based on similar clones that have similar but not identical sets of identifiers. They found 87 defects when analyzing 955 suspicious clones. Bakota and colleagues [3] detected six defects based on their analysis of 60 change anomalies in the evolution of clones. Clones in Linux and Eclipse have been analyzed by Jiang and colleagues [14]. In addition to the clone itself, they analyzed each clone’s context to find defects. Out of 1342 spurious clones and their context, they detected 62 defects. Juergens and his colleagues [17] (with the help of system experts) checked clones that are similar but not identical for defects. Out of 1427 candidates, 107 were found to contain defects of different severity.

The studies show that at least some clones facilitate defects. Nevertheless, all studies except for the one by Bakota and his colleagues analyzed clones in a single version. It is thus possible that the defects were caused by something else than an inconsistent change.

4. EXTRACTING GENEALOGIES

This section explains how we extract the clone evolution graph from the history of a system. The process is based on our incremental clone detection algorithm for efficiently detecting clones in multiple versions of a system [9, 10]. The algorithm processes versions of the system one after another. For each version, the following steps are performed:

1. Process changed files and update data structures.
2. Detect clones for the current version.
3. Map clones between previous and current version.

These steps are repeated for every version until no more versions are available.

4.1 Processing Changed Files

Processing changed files is done as in the algorithm’s original version. We distinguish three primary ways in which a file can be changed: addition, deletion, and modification. Apart from that, the algorithm also supports renaming of files. Which files have been changed or renamed is given as input to our algorithm. The algorithm internally uses a suffix tree to detect duplication. For details on how changed files are processed and the suffix tree is updated, please refer to [9, 10].

When a modified file is processed, we calculate the difference between the file’s old and new version. This information is needed later when fragments are mapped. We use our implementation of Myers’ algorithm [29] which we adopted to work for token sequences.

4.2 Detecting Clones

In this phase, the clones that exist in the current version of the system are detected using a two-step process. First, type-1 clones are extracted from the suffix tree. We deliberately do not extract type-2 clones at this point, because this yields repetitive structures with a high ratio of identifiers which are primarily irrelevant clones. By first extracting only type-1 clones, we ensure that there are identical parts of sufficient length, before allowing any differences. In the second step, neighboring type-1 clones are merged to larger clones of types 2 and 3.

We use a simple algorithm inspired by Baker’s approach [2] to extract type-1 clone classes (identical fragments) from the suffix tree. To exclude irrelevant clone classes (for example, where each fragment is a single token), we apply a minimum length that fragments are required to have. Note that this is not the minimum length for the final clones as specified by the user.

Detection of type-2 and type-3 clones is done by merging neighboring type-1 clones and inspired by previous work by Ueda and colleagues [34]. The type-1 clone classes that have been extracted from the updated GST are first expanded to type-1 clone pairs. These pairs are sorted into buckets according to the two files that contain the fragments. Each bucket is sorted based on the location of the pairs’ fragments.

Each bucket is processed separately. The pairs are processed in sequence. For each pair, the algorithm checks whether it can extend one of the previous pairs. Two fragments are close enough, if the distance d between the fragments equals or is less than the shorter of the two fragments. This is driven by the assumption that the differences may not be larger than the similarities. In practice, we have observed that the differences are always much less than the similarities if pairs are merged using this rule. If the corresponding fragments of both pairs are close enough, the clone pairs can be merged.

We provide an example in Figure 2 with two files A and B and three clone pairs a, b, c each of which has one fragment
in each file. Assume that the pairs are processed \(c, b, a\). The first pair \(c\) cannot extend any pair, because it is the first one to be processed. Pair \(b\) cannot extend \(c\), because the fragments are twisted, that is, in file \(A\), \(c\)'s fragment is located before \(b\)'s fragment whereas in file \(B\), \(c\)'s fragment is located after \(b\)'s fragment. Nevertheless, clone pair \(a\) can extend clone pair \(b\), because the gaps are one and two tokens long and the shortest fragment is two tokens long.

Using the described procedure, our algorithm establishes chains of clone pairs that can be merged into a single but larger clone pair. That is, the algorithm merges an arbitrary number of clone pairs into a single pair obeying a user-specified minimum length. The algorithm has worst case quadratic complexity with respect to the number of identical pairs. Nevertheless, the number of identical pairs is usually small and we exploit the sorting to eliminate unnecessary calculations where possible.

After the gapped clone pairs have been created, they are grouped again into clone classes according to their similarities. This is done based on the common subsequence of the pair’s fragments. For a gapped clone pair that has been merged from smaller type-1 clone pairs, the common subsequence is the concatenated token sequence of the type-1 pairs. We combine the fragments of two pairs into a single clone class when the common subsequences of both pairs are identical. Finally, we check the type of each merged clone class. If the gaps—the non-identical parts—between the identical pairs. Nevertheless, the number of identical pairs is usually small and we exploit the sorting to eliminate unnecessary calculations where possible.

After the gapped clone pairs have been created, they are grouped again into clone classes according to their similarities. This is done based on the common subsequence of the pair’s fragments. For a gapped clone pair that has been merged from smaller type-1 clone pairs, the common subsequence is the concatenated token sequence of the type-1 pairs. We combine the fragments of two pairs into a single clone class when the common subsequences of both pairs are identical. Finally, we check the type of each merged clone class. If the gaps—the non-identical parts—between the fragments consist of only identifiers and literals, the clone class is of type 2, otherwise of type 3.

The final set of clones contains type-1 clone classes (the ones initially extracted), as well as type-2 and type-3 clone classes that have been merged. We then discard all type-1 clone classes whose fragments are shorter than a user-specified minimum length. This cannot be done earlier because we required the shorter type-1 classes for merging. Furthermore, we exclude all type-1 clone classes whose fragments are completely contained in the fragments of another clone class of type 2 or 3 (this can happen when the fragments of the type-1 clone class have been merged).

### 4.3 Mapping Clones

The goal of mapping fragments is to find for each fragment in the current version its ancestors in the previous version. Mapping fragments is relatively simple for files that are added or remain unchanged. All fragments in an added file have no ancestors. Note that duplicated files are also treated as added. For unchanged files, each fragment has exactly one ancestor which is found at exactly the same location.

For modified files, we exploit the difference information which has been calculated when changed files have been processed. For each fragment of the system’s previous version that is contained in a modified file, we calculate its occurrence in the current version using the difference information being a list containing information about added or deleted token chunks. The occurrence is calculated by processing all modifications that affect the fragment. If tokens are added or deleted before the fragment, its location is adjusted accordingly. If tokens are removed or added within the fragments bounds, the fragment becomes shorter or longer, respectively. After the relevant modifications have been processed, the calculated occurrence of the fragment describes how the fragment “would look like” in the current version if it still existed.

After calculating a fragment’s hypothetical occurrence, the algorithm checks whether there exists a cloned fragment in the system’s current version that equals the calculated occurrence. The check can be done in constant time using a hash value computed from the fragments’ locations. If such a fragment is found, the fragment from the previous version is added to the set of ancestors of the fragment in the current version.

Note that our mapping technique is independent from the source of the difference information. We currently use a standard differencing algorithm, but the information might also come from other sources, for example an IDE.

### 5. STUDY SETUP

This section describes the setup that we used to collect the data we need to answer the research questions. The setup includes our choice of subject systems, parameters for our clone detection tool, and an overview of our procedure to collect relevant data.

#### 5.1 Subject Systems

We used the following three subject systems for our study.

- Our first subject system is our token-based clone detection tool, CLONES, itself. CLONES is written in Ada and part of the Bauhaus project\(^1\). We included this system, because our knowledge of the code allows us to much better assess clones and changes to clones in the code.

- The second system is the web server HTTPD.\(^2\) HTTPD is part of the Apache project and written in C.

- Our third subject system is JabRef.\(^3\) JabRef is a bibliography manager written in Java.

We chose these systems, because they have been used in some previous studies including ours \[8\]. All systems are mature and have a reasonably long history. Each system has had a multitude of developers contributing and is of reasonable size. This mitigates the threat that a system is more or less maintained by a single developer who is an expert regarding every part of the system. For CLONES and JabRef, 15 distinct developers contributed changes relevant to our study. For HTTPD, 45 developers were involved in maintenance.

When analyzing clone evolution based on multiple versions of a system’s source code, the interval between each

---

\(^1\)http://www.bauhaus-stuttgart.de

\(^2\)http://httpd.apache.org

\(^3\)http://jabref.sourceforge.net
pair of consecutive versions is of great importance. Increasing the interval between versions leads to multiple changes being merged into a single change and consequently results in loss of detail. To capture data with the highest degree of detail that is available, we consider each individual commit to the system's repository as a version. We excluded commits that did not change source files relevant to our analysis.

In addition to the interval between consecutive versions, we require a first and a last version that frame the period of the system's lifetime that we analyze. The number of versions that we can analyze is limited by our hardware. We chose to analyze all projects over the period of five years, starting from the very first version in 2005 to the very last version in 2009. Although more versions could have been analyzed for smaller projects, we chose the same scope for all subject systems to make the results comparable. Table 1 summarizes our subject systems and details about the versions we have analyzed.

### 5.2 Clone Detection Parameters

This section describes the parameters that we used to extract the relevant clone evolution data.

**Exclusion of header files.** For this study, we were interested in clones and changes that affect the systems' logic. We excluded all header files (*.h) from httpd and all Ada specification files (*.ads) from CLONES.

**Exclusion of generated code.** Generated code is not maintained by hand and thus, most problems caused by clones do not apply to generated code. Especially consistent and inconsistent changes to automatically generated files are not relevant to our study. We therefore exclude all generated files from our analysis. For CLONES all generated files are kept in a single directory which we excluded. Regarding the other systems, we identified generated files by searching directory and file names as well as comments inside a file for indications.

**Method Separation.** Due to a token-based clone detector's missing knowledge about syntactic structures, clones may embrace more than a single method or function in the source code, that is, clones may start in one method and continue into the next method. To eliminate these artifacts, we preprocess the token sequence of each file before clone detection. We use regular expressions to identify the boundaries of methods and functions and insert unique sentinel tokens that can never be part of a clone. This procedure ensures that clones do not cross method boundaries. For httpd, we used a heuristic to identify the boundaries in the presence of preprocessor directives.

**Token Exclusion.** Another source of less relevant clones are reoccurring patterns imposed by the programming language. Like method separation, this step preprocesses each file's token sequence before clones are detected. This step excludes all `#include`-directives in the C code, `with-stmt`-statements in the Ada code, and `import`-statements in the Java code.

In addition to the interval between consecutive versions, we require a first and a last version that frame the period of the system's lifetime that we analyze. The number of versions that we can analyze is limited by our hardware. We chose to analyze all projects over the period of five years, starting from the very first version in 2005 to the very last version in 2009. Although more versions could have been analyzed for smaller projects, we chose the same scope for all subject systems to make the results comparable. Table 1 summarizes our subject systems and details about the versions we have analyzed.

#### 5.2 Clone Detection Parameters

This section describes the parameters that we used to extract the relevant clone evolution data.

**Exclusion of header files.** For this study, we were interested in clones and changes that affect the systems' logic. We excluded all header files (*.h) from httpd and all Ada specification files (*.ads) from CLONES.

**Exclusion of generated code.** Generated code is not maintained by hand and thus, most problems caused by clones do not apply to generated code. Especially consistent and inconsistent changes to automatically generated files are not relevant to our study. We therefore exclude all generated files from our analysis. For CLONES all generated files are kept in a single directory which we excluded. Regarding the other systems, we identified generated files by searching directory and file names as well as comments inside a file for indications.

**Method Separation.** Due to a token-based clone detector's missing knowledge about syntactic structures, clones may embrace more than a single method or function in the source code, that is, clones may start in one method and continue into the next method. To eliminate these artifacts, we preprocess the token sequence of each file before clone detection. We use regular expressions to identify the boundaries of methods and functions and insert unique sentinel tokens that can never be part of a clone. This procedure ensures that clones do not cross method boundaries. For httpd, we used a heuristic to identify the boundaries in the presence of preprocessor directives.

**Token Exclusion.** Another source of less relevant clones are reoccurring patterns imposed by the programming language. Like method separation, this step preprocesses each file's token sequence before clones are detected. This step excludes all `#include`-directives in the C code, `with-stmt`-statements in the Ada code, and `import`-statements in the Java code.

#### 5.2 Clone Detection Parameters

This section describes the parameters that we used to extract the relevant clone evolution data.

**Exclusion of header files.** For this study, we were interested in clones and changes that affect the systems' logic. We excluded all header files (*.h) from httpd and all Ada specification files (*.ads) from CLONES.

**Exclusion of generated code.** Generated code is not maintained by hand and thus, most problems caused by clones do not apply to generated code. Especially consistent and inconsistent changes to automatically generated files are not relevant to our study. We therefore exclude all generated files from our analysis. For CLONES all generated files are kept in a single directory which we excluded. Regarding the other systems, we identified generated files by searching directory and file names as well as comments inside a file for indications.

**Method Separation.** Due to a token-based clone detector's missing knowledge about syntactic structures, clones may embrace more than a single method or function in the source code, that is, clones may start in one method and continue into the next method. To eliminate these artifacts, we preprocess the token sequence of each file before clone detection. We use regular expressions to identify the boundaries of methods and functions and insert unique sentinel tokens that can never be part of a clone. This procedure ensures that clones do not cross method boundaries. For httpd, we used a heuristic to identify the boundaries in the presence of preprocessor directives.

**Token Exclusion.** Another source of less relevant clones are reoccurring patterns imposed by the programming language. Like method separation, this step preprocesses each file's token sequence before clones are detected. This step excludes all `#include`-directives in the C code, `with-stmt`-statements in the Ada code, and `import`-statements in the Java code.

#### 5.2 Clone Detection Parameters

This section describes the parameters that we used to extract the relevant clone evolution data.

**Exclusion of header files.** For this study, we were interested in clones and changes that affect the systems' logic. We excluded all header files (*.h) from httpd and all Ada specification files (*.ads) from CLONES.

**Exclusion of generated code.** Generated code is not maintained by hand and thus, most problems caused by clones do not apply to generated code. Especially consistent and inconsistent changes to automatically generated files are not relevant to our study. We therefore exclude all generated files from our analysis. For CLONES all generated files are kept in a single directory which we excluded. Regarding the other systems, we identified generated files by searching directory and file names as well as comments inside a file for indications.

**Method Separation.** Due to a token-based clone detector's missing knowledge about syntactic structures, clones may embrace more than a single method or function in the source code, that is, clones may start in one method and continue into the next method. To eliminate these artifacts, we preprocess the token sequence of each file before clone detection. We use regular expressions to identify the boundaries of methods and functions and insert unique sentinel tokens that can never be part of a clone. This procedure ensures that clones do not cross method boundaries. For httpd, we used a heuristic to identify the boundaries in the presence of preprocessor directives.

**Token Exclusion.** Another source of less relevant clones are reoccurring patterns imposed by the programming language. Like method separation, this step preprocesses each file's token sequence before clones are detected. This step excludes all `#include`-directives in the C code, `with-stmt`-statements in the Ada code, and `import`-statements in the Java code.

#### 5.3 Procedure

This section describes the overall procedure we used during our analysis. The process is illustrated in Figure 3.

1. We extracted the relevant versions of each subject system from its SUBVERSION repository and stored them locally.
2. This step is necessary since fast access to the source code—for detection and analysis—is of great advantage. It allows to repeatedly access the source code without the overhead of accessing a distant repository. Note that we store only changed source code for each version, keeping the space requirement low. This step had to be done only once for each system, whereas every following step had to be done for each combination of parameters.
3. We used our incremental detection algorithm IDA to extract the clone evolution data as described in Section 4.

#### 6. RESULTS

This section presents the results of our case study and describes the measurements we used to answer our research questions. We first quantified the relevant data to provide an overall impression of the cloning situation in our subject systems.
For each subject system and each possible combination of clone type and minimum length, we calculated the total number of clone classes aggregated over all versions, the number of genealogies that were obtained from linking the clone classes, and the number of situations in which a clone class was changed. Consider our example of an evolution graph in Figure 1. That graph contains 18 clone classes which are grouped into four distinct genealogies based on the ancestry of their fragments. There are six situations in which clone classes have been changed, indicated by the shaded clone classes.

The corresponding quantities for our subject systems are given in Table 2. Not surprisingly, increasing the minimum clone length reduces the number of detected clones for each system. When searching for all types of clones, more clones are detected compared to only looking for type-1 clones. For every system, the number of genealogies is proportional to the number of detected clones. Nevertheless, the number of clone classes for each genealogy differs between the systems. In CLONES, each genealogy embraces roughly 100 clone classes, in httpd 1000 clone classes, and in JabRef 200 clone classes. Although we have analyzed only twice as many versions of httpd than of the other systems, the number of clone classes per genealogy is ten (five) times higher in httpd. This may be due to clones in httpd changing less frequently compared to the other two systems. If clones change less, genealogies are more stable, that is, new genealogies appear only rarely and existing genealogies disappear only rarely. Consequently, genealogies contain more clone classes due to their longer life expectancy. This is supported by the number of changes displayed in the last column of Table 2. The number of situations in which clones were changed is not significantly higher for httpd compared to the other systems—although we have analyzed more than twice as many versions for httpd and the number of detected clone classes is significantly higher.

In summary, the cloning situation is quite different for our subject systems. CLONES contains comparatively few clones which are, however, changed more frequent. httpd has the highest clone density but clones are changed less frequently. JabRef falls in-between those two.

6.1 Change Propagation

We now use the information about clone genealogies and changes to clones to answer Question 1 and estimate the additional costs of change propagation caused by clones. We assume that clones may cause additional change effort only when they are modified. Hence, clones that are never changed can never induce costs for change propagation.

To analyze how often clones are changed during their lifetime, we counted the number of changes for each genealogy. Let the change frequency (cf) of a genealogy be the number of the genealogy’s clone classes marked as changed. Change frequencies are also given for the genealogies in our example shown in Figure 1. The number of genealogies and its percentage compared to all genealogies are shown in Table 3 for each change frequency. We chose to include these detailed results because the choice of clone detection parameters influences the results and different use cases may require different parameter settings. If furthermore allows to estimate the impact of changing a given parameter. The number of genealogies with a change frequency higher than 5 have been combined due to space limitation. We provide averages at the bottom of the table. The * represents the combination of all values.

Never changed. We first look at the clones that never changed. The column labeled “0” shows how many genealogies were never affected by change during their lifetime. The next column to the right gives the percentage when compared to the number of all genealogies. The values indicate that for httpd and JabRef, more than half of the clone genealogies were never changed during their lifetime. These numbers tell us that roughly half of the clones have—under no circumstances—induced costs for change propagation. For CLONES, 35.1% of the genealogies were never changed. This may be due to the high number of changes—considering CLONES’ size and the number of versions analyzed—compared to the other systems. Taking the average of all measurements, 47.5% of all genealogies were never changed.

The numbers are almost the same, independent of whether we consider only type-1 clones or clones of all types. Increasing the minimum length of clones reduces the percentage of genealogies that are never changed. We think this is due to longer clones being more susceptible to change. When the smaller clones are not considered, the remaining longer ones are more likely to be affected by changes.

Changed once. Apart from the genealogies that never changed, the genealogies that changed only once are of interest to us. During the lifetime of these genealogies, there has been a single change applied to one clone class of the genealogy. Assuming that removing a clone requires changing it at least once, genealogies that were changed only once did not cause additional costs compared to when the clone would have been removed. It is, however, debatable, whether removing a clone or performing an arbitrary change to it requires more effort.

In CLONES 46.6%, in httpd 38.7%, and in JabRef 35.6% of the genealogies were changed exactly once. For CLONES the number of genealogies changed once is higher than the number of genealogies that never changed. For the other two systems, more genealogies were never changed than changed.

\[^{4}\text{We refer to the act of changing the source code and exclude the effort to understand or test the code.}\]
once. The overall average of genealogies that changed exactly once is 40.3%, being almost as high as the percentage of clones that never changed.

The type of clones has only a minor impact on the results. When considering only type-1 clones, 41.2% of the genealogies were changed exactly once. Interestingly, this is the inverse compared to the genealogies that never change, where the percentage of genealogies decreased as the minimum length of clones increased.

### Changed more than once

The remaining category are clone genealogies that were changed more than once. Table 3 lists the number of genealogies for change frequencies from 2 to 5 and the aggregated number of genealogies that changed more than five times. In general, the number of genealogies decreases as the change frequency increases. While there are still some genealogies that changed twice (7.5%) or thrice (2.7%), higher change frequencies are only scattered. Clone genealogies that change multiple times are likely to increase the change effort and should thus be regarded as harmful clones. These clones should be inspected and removed where possible.

<table>
<thead>
<tr>
<th>System</th>
<th>Type</th>
<th>Min.</th>
<th>#Geneal.</th>
<th>0 [%]</th>
<th>1 [%]</th>
<th>2 [%]</th>
<th>3 [%]</th>
<th>4 [%]</th>
<th>5 [%]</th>
<th>&gt;5 [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONES</td>
<td>1</td>
<td>30</td>
<td>876</td>
<td>385</td>
<td>43.9</td>
<td>356</td>
<td>40.6</td>
<td>84</td>
<td>9.6</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>50</td>
<td>317</td>
<td>109</td>
<td>34.4</td>
<td>154</td>
<td>48.6</td>
<td>34</td>
<td>10.7</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>100</td>
<td>56</td>
<td>86</td>
<td>27.9</td>
<td>46</td>
<td>53.5</td>
<td>10</td>
<td>11.6</td>
<td>4</td>
</tr>
<tr>
<td>1-3</td>
<td>30</td>
<td>1,027</td>
<td>434</td>
<td>42.3</td>
<td>429</td>
<td>41.8</td>
<td>89</td>
<td>8.69</td>
<td>40</td>
<td>3.9</td>
</tr>
<tr>
<td>1-3</td>
<td>50</td>
<td>523</td>
<td>179</td>
<td>34.2</td>
<td>247</td>
<td>47.2</td>
<td>51</td>
<td>9.8</td>
<td>26</td>
<td>5.0</td>
</tr>
<tr>
<td>1-3</td>
<td>100</td>
<td>189</td>
<td>53</td>
<td>28.0</td>
<td>90</td>
<td>47.6</td>
<td>22</td>
<td>11.6</td>
<td>12</td>
<td>6.3</td>
</tr>
<tr>
<td>HTTPD</td>
<td>1</td>
<td>30</td>
<td>1,532</td>
<td>961</td>
<td>62.7</td>
<td>497</td>
<td>32.4</td>
<td>54</td>
<td>3.5</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>50</td>
<td>742</td>
<td>421</td>
<td>56.7</td>
<td>275</td>
<td>37.1</td>
<td>33</td>
<td>4.4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>100</td>
<td>245</td>
<td>109</td>
<td>44.5</td>
<td>116</td>
<td>47.3</td>
<td>12</td>
<td>4.9</td>
<td>5</td>
</tr>
<tr>
<td>1-3</td>
<td>30</td>
<td>1,902</td>
<td>1,134</td>
<td>59.6</td>
<td>634</td>
<td>33.3</td>
<td>82</td>
<td>4.3</td>
<td>35</td>
<td>1.8</td>
</tr>
<tr>
<td>1-3</td>
<td>50</td>
<td>1,145</td>
<td>609</td>
<td>53.2</td>
<td>430</td>
<td>37.6</td>
<td>59</td>
<td>5.2</td>
<td>33</td>
<td>2.9</td>
</tr>
<tr>
<td>1-3</td>
<td>100</td>
<td>461</td>
<td>196</td>
<td>42.5</td>
<td>205</td>
<td>44.5</td>
<td>34</td>
<td>7.4</td>
<td>17</td>
<td>3.7</td>
</tr>
<tr>
<td>JARRef</td>
<td>1</td>
<td>30</td>
<td>927</td>
<td>515</td>
<td>55.6</td>
<td>321</td>
<td>34.6</td>
<td>63</td>
<td>6.8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>50</td>
<td>416</td>
<td>221</td>
<td>53.1</td>
<td>160</td>
<td>38.5</td>
<td>25</td>
<td>6.0</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>100</td>
<td>132</td>
<td>72</td>
<td>54.5</td>
<td>51</td>
<td>38.6</td>
<td>7</td>
<td>5.3</td>
<td>1</td>
</tr>
<tr>
<td>1-3</td>
<td>30</td>
<td>1,220</td>
<td>675</td>
<td>55.3</td>
<td>410</td>
<td>33.6</td>
<td>90</td>
<td>7.4</td>
<td>27</td>
<td>2.2</td>
</tr>
<tr>
<td>1-3</td>
<td>50</td>
<td>731</td>
<td>386</td>
<td>52.8</td>
<td>264</td>
<td>36.1</td>
<td>56</td>
<td>7.7</td>
<td>13</td>
<td>1.8</td>
</tr>
<tr>
<td>1-3</td>
<td>100</td>
<td>269</td>
<td>147</td>
<td>54.6</td>
<td>87</td>
<td>32.3</td>
<td>25</td>
<td>9.3</td>
<td>1.1</td>
<td>4</td>
</tr>
<tr>
<td>CLONES</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>35.1</td>
<td>–</td>
<td>46.6</td>
<td>–</td>
<td>10.3</td>
<td>–</td>
<td>4.7</td>
</tr>
<tr>
<td>HTTPD</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>53.2</td>
<td>–</td>
<td>38.7</td>
<td>–</td>
<td>5.0</td>
<td>–</td>
<td>2.1</td>
</tr>
<tr>
<td>JARRef</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>54.3</td>
<td>–</td>
<td>35.6</td>
<td>–</td>
<td>7.1</td>
<td>–</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>1</td>
<td>–</td>
<td>48.1</td>
<td>–</td>
<td>41.2</td>
<td>–</td>
<td>7.0</td>
<td>–</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>1-3</td>
<td>–</td>
<td>46.9</td>
<td>–</td>
<td>39.3</td>
<td>–</td>
<td>7.9</td>
<td>–</td>
<td>3.2</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>30</td>
<td>–</td>
<td>53.2</td>
<td>–</td>
<td>36.1</td>
<td>–</td>
<td>6.7</td>
<td>–</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>50</td>
<td>–</td>
<td>47.4</td>
<td>–</td>
<td>40.9</td>
<td>–</td>
<td>7.3</td>
<td>–</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>100</td>
<td>–</td>
<td>42.0</td>
<td>–</td>
<td>44.0</td>
<td>–</td>
<td>8.4</td>
<td>–</td>
<td>3.1</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>–</td>
<td>47.5</td>
<td>40.3</td>
<td>7.5</td>
<td>2.7</td>
<td>0.9</td>
<td>0.5</td>
<td>0.8</td>
<td>0.8</td>
</tr>
</tbody>
</table>

### Table 3: Change frequency of clone genealogies

<table>
<thead>
<tr>
<th>Change frequency</th>
<th>0 [%]</th>
<th>1 [%]</th>
<th>2 [%]</th>
<th>3 [%]</th>
<th>4 [%]</th>
<th>5 [%]</th>
<th>&gt;5 [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>42.0</td>
<td>44.0</td>
<td>8.4</td>
<td>3.1</td>
<td>1.0</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>50</td>
<td>47.4</td>
<td>40.9</td>
<td>7.3</td>
<td>2.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>30</td>
<td>46.9</td>
<td>39.3</td>
<td>7.9</td>
<td>3.2</td>
<td>1.1</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>1-3</td>
<td>53.2</td>
<td>36.1</td>
<td>6.7</td>
<td>2.3</td>
<td>0.8</td>
<td>0.5</td>
<td>0.4</td>
</tr>
<tr>
<td>1-3</td>
<td>47.4</td>
<td>40.9</td>
<td>7.3</td>
<td>2.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>1-3</td>
<td>42.0</td>
<td>44.0</td>
<td>8.4</td>
<td>3.1</td>
<td>1.0</td>
<td>0.8</td>
<td>0.8</td>
</tr>
</tbody>
</table>

### 6.2 Unintentional Inconsistencies

Apart from increasing the change effort, clones are also said to facilitate problems caused by unintentional inconsistent changes. To explore this assumption and to answer Question 2, we analyzed the changes to clones in detail. Unfortunately, manual assessment requires significant effort and is always threatened to be biased. From our experience we know, that assessing the impact of changes to code we are not familiar with is hardly possible for the majority of changes. To ensure familiarity with the code and its history, we restricted this part of our study to the system CLONES. Consequently, we are in a much better position to judge the changes and to explain our findings than if we were analyzing an unfamiliar system. The disadvantage of studying clones is, that some of its developers were aware of clone-related problems (see our threats to validity in Section 6.3).

For assessment, we used only a single parameter setting to keep the assessment effort manageable. We chose to look at changes to type-1 clones only, because having identical fragments before the change makes the evaluation of the change itself much easier. For the minimum clone length, we chose 50 tokens. This setting provided us with a total number of genealogies that were changed more than once. Increasing the minimum length of clones decreases the percentage of genealogies that are never changed while increasing the percentage of all genealogies that changed at least once. We believe that this is caused by the increased likelihood of larger clones being affected by change. Excluding clones of shorter length—most of which are not changed—leaves only the longer clones that are more susceptible to change due to their increased length.
function Is_Right_Extensible
(A, B : Token_Table_Index;
Len : Natural)
return Boolean
is ...
(a) Version i

function Is_Right_Extensible
(A : in Token_Table_Index;
B : in Token_Table_Index;
Len : in Natural)
return Boolean
is ...
(b) Version i + 1

Figure 4: Semantic-preserving change

304 changes to clone classes. To answer Question 2, every change was analyzed to decide whether it contains an unintentional inconsistency—that is, a missing propagation of the change. We further assign each unwanted inconsistency one of two severity levels:

**Low**: The change *should have been consistent*. It does not cause the program to crash or produce wrong results, but may complicate future maintenance and facilitate further inconsistencies.

**High**: The change *must have been consistent*. The inconsistency may cause the program to crash or produce wrong results.

The following paragraphs summarize the results of our analysis of changes to clones.

**Consistent changes.** Our objective is to identify unintentional inconsistent changes within the set of all changes. Consequently, we first removed all consistent changes—that is, changes where exactly the same modifications have been done to each fragment of the clone class. In total, 173 (56.9%) of the changes were consistent. The remaining 131 inconsistent changes were inspected manually.

**Semantic-preserving.** Inconsistent changes are detected based on the changes to the token sequences of the respective fragments. However, not every change to the token sequence changes the semantic of the program. For example, the rewriting of the parameter list as shown in Figure 4 does not change the semantic (in is the default mode for parameters in Ada), but may still be responsible for an inconsistent change. If semantic-preserving changes are unwanted inconsistencies, their severity can be only low.

We manually checked each inconsistent change whether it altered the semantic of the program or not. We found that almost half (51 out of 131) of the inconsistent changes did not change the semantic of the program. Many of the inconsistencies resulted from unused variable declarations or unnecessary uses of fully qualified names being removed.

**Unwanted Inconsistencies.** The final and most important assessment for each change is whether it features a missing propagation and if it does, whether the severity is low or high. An example for an inconsistency with high severity is shown in Figure 5. The missing check to prevent Ceiling Prime being called with 0 (the type **Positive**

<table>
<thead>
<tr>
<th>Table 4: Classification of changes in clones</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
</tr>
<tr>
<td>Consistent</td>
</tr>
<tr>
<td>Inconsistent</td>
</tr>
<tr>
<td>Semantic-preserving</td>
</tr>
<tr>
<td>Unintentional</td>
</tr>
<tr>
<td>Low severity</td>
</tr>
<tr>
<td>High severity</td>
</tr>
</tbody>
</table>

expects numbers greater than 0) is inserted in only one of the fragments making this an incomplete bug removal. The change has high severity, because the change must be propagated to all fragments. Unless the missing check is inserted in all fragments, the program may crash when encountering empty token sequences.

Analyzing the 131 inconsistent changes, we found that 45 of them were unintentionally inconsistent. This amounts to 14.8% of the total number of changes to clones. Regarding severity, four out of five unwanted inconsistencies had low severity—for example changes to debugging code. Of all 36 changes classified as low, 20 are also semantic-preserving. In total, only 9 inconsistencies with high severity were found. This amounts to 3% of all changes. Table 4 summarizes our results regarding the classification of changes. Note that changes may belong to multiple categories.

### 6.3 Threats to Validity

There are certain threats to the validity of our results.

**Clone detection parameters.** As with most studies on clones, the parameters used for detecting clones have a major impact on the results. We deliberately used different combinations of parameters to analyze their impact. This included the type of clones that were detected and the minimum length that clones were required to have. We presented detailed results in Table 3 and described the impact of the parameters.

**Precision.** The precision of a clone detection tool influences the validity of the results. Although token-based clone detection has—in general—lower precision than approaches using abstract syntax trees or graphs, we carefully selected our parameters to exclude irrelevant clones from the results. Furthermore, we used a novel approach that requires clones to have identical parts of sufficient length to detect type-2 and type-3 clones—a frequent source of irrelevant clones using traditional approaches. One of the authors manually inspected a sample of clones to ensure that detected clones are relevant. The only clones we found to be less relevant are those in handling of command line parameters.

**Study period.** The part of a system’s history that we used for our study also has an impact on the result. For every subject system, we have analyzed five years of its history to reduce the impact of special phases (e.g., major refactorings). Using only a part of a system’s history also crops genealogies, that is, genealogies might have existed before the beginning of the study period and might exist beyond the end of the study period. When considering only genealogies that appear after the first version and disappear before the last version of the study period, the majority of genealogies has a change frequency of 1 (instead of 0). Nevertheless, the percentage of genealogies changed more than once remains
almost the same. Consequently, the ratio between genealogies changed never or once and genealogies changed more than once remains also roughly the same.

**Mapping.** The mapping between fragments of consecutive versions is another potential threat to the validity. As genealogies are created based on the mapping of fragments, a missing connection will break genealogies apart, leaving two genealogies each of which has a change frequency equal or less to the combined genealogy. The reason for an incomplete mapping can be the algorithm to calculate differences between token sequences—which has to deal with ambiguous situations—and clone classes becoming left-extensible or right-extensible due to changes or removal of fragments. However, according to our experience from previous studies and careful analysis of the clone evolution data, we have found that these situations are very rare.

**Analyzing clones.** Another threat may arise from the analysis of our own program CLONES. Although 15 different developers have contributed to CLONES, most of them knew about clone-related problems—not saying they were aware of them during development. This might have an impact on the number of unintentional inconsistencies. Thus, the percentage of unwanted inconsistencies might be higher for projects where developers are completely unaware of clones.

**Manual assessment.** Manual assessment of changes to clones depends on the judges and their expertise. We have deliberately chosen a system we are familiar with to reliably judge the impact of changes. Furthermore, we applied clear assessment criteria that reduced ambivalent decisions.

7. DISCUSSION

Clones are said to increase the change effort when software is maintained, because changes have to be propagated to all clones. However, looking at our results we have presented in Section 6.1, we found that almost half of the clones have never been changed during their lifetime. These clones have never caused additional costs for change propagation. Apart from the clones that never changed, another 40.3% changed only once during their lifetime.

We assume that removal is the only option to prevent additional change effort since semi-automated change propagation (e.g., [33]) still requires effort for manual verification of the suggested propagation. We also assume that the additional effort caused by a single change to a clone class is less than the effort to remove the respective clones by refactoring. That means, for 87.8% of the clones (the ones that never changed or changed once), removal would not have been beneficial because the effort to remove the clones would have been larger than the additional change effort that would have been prevented. Only for the remaining 12.2% of clones that changed more than once, removal might have been beneficial. Furthermore, we counted all changes including those that are meant to be independent—that is, they do not need to be propagated to the other clones of the class. Hence, the 87.8% is an under-estimation of the true percentage.

Our results show that the major percentage of clones does not cause additional effort, independent of whether the clone is “structurally interesting” or not. Based on our findings, we suggest to consult change information when deciding which clones are relevant and which clones are not relevant. Although the data describe only the past evolution of clones, we believe it provides important information to predict their future evolution.

When a cloned fragment is changed, the change might need to be propagated to the other copies. If this is not done—or not done correctly—new problems may arise or existing problems might not be completely mitigated. We analyzed changes to clones in our own code and found that 14.8% of all changes were unintentionally inconsistent. Re-analyzed changes to clones in our own code and found that almost half of the clones have high severity.

In the past evolution of clones, we found that almost half of the clones have never been changed during their lifetime. These clones have never caused additional costs for change propagation. Apart from the clones that never changed, another 40.3% changed only once during their lifetime.

We assume that removal is the only option to prevent additional change effort since semi-automated change propagation (e.g., [33]) still requires effort for manual verification of the suggested propagation. We also assume that the additional effort caused by a single change to a clone class is less than the effort to remove the respective clones by refactoring. That means, for 87.8% of the clones (the ones that never changed or changed once), removal would not have been beneficial because the effort to remove the clones would have been larger than the additional change effort that would have been prevented. Only for the remaining 12.2% of clones that changed more than once, removal might have been beneficial. Furthermore, we counted all changes including those that are meant to be independent—that is, they do not need to be propagated to the other clones of the class. Hence, the 87.8% is an under-estimation of the true percentage.

Our results show that the major percentage of clones does not cause additional effort, independent of whether the clone is “structurally interesting” or not. Based on our findings, we suggest to consult change information when deciding which clones are relevant and which clones are not relevant. Although the data describe only the past evolution of clones, we believe it provides important information to predict their future evolution.

When a cloned fragment is changed, the change might need to be propagated to the other copies. If this is not done—or not done correctly—new problems may arise or existing problems might not be completely mitigated. We analyzed changes to clones in our own code and found that 14.8% of all changes were unintentionally inconsistent. Regarding severity, 11.8% of the unwanted inconsistencies had low severity whereas only 9 (3.0%) of all changes were found to have high severity.

On the one hand, we can confirm the results of previous studies [3, 14, 17, 25] that identified faults based on inconsistencies between clones or their contexts. There are clones that facilitate problems when being changed. On the other hand, our results show that 85.2% of the changes are consistent or intentionally inconsistent confirming findings of previous studies [1, 32]. This is important for developing clone-management tools, because too many false alarms regarding potential problems caused by inconsistent changes may be counterproductive for the adoption of clone management in practice.

![Figure 5: Unintentional inconsistency with high severity](image-url)

```
1 function Ceiling_Prime (Number : in Positive) return Positive is
2 begin
3 Table_Size := Natural := Natural (Float (Number_OF_Tokens) * 2.4);
4 if Table_Size > 0 then
5 Table_Size := Ceiling_Prime (Table_Size);
6 end if;
7 end Initialize;
```

(a) Version \( i \)

(b) Version \( i + 1 \)
8. CONCLUSION

Code clones are said to cause a variety of problems among which are additional costs for change propagation and the risk of unintentional inconsistencies. We have analyzed clone evolution using the history of three subject systems to estimate the threat potential of clones. Regarding the additional change effort, the answer to Question 1 is that 87.8% of clones were never changed or changed only once during their lifetime. Thus, we cannot say that these clones caused additional change effort (assuming the effort to remove them equals the effort to change them once).

Regarding Question 2, the answer is that only 14.8% of all changes to clones are unintentionally inconsistent. Furthermore, only few of these changes have high severity, while the others are more or less cosmetic.

Please note that our study focused on two prominent clone-related threats. Apart from these, there are other problems caused by the presences of code clones. These include, for example, increased system size and additional effort to read and understand the code. Hence, our results are not to be regarded as a general acquittal for clones.

The implication of our results is, that the history of clones provides important information to determine their relevance and threat potential. This is important regarding the increasing number of clone management tools. To successfully establish these tools in the daily maintenance practice, it is important to reliably identify the clones with high threat potential and not besiege maintainers with false alarms.

9. REFERENCES