Clone Smells in Software Evolution

Tibor Bakota, Rudolf Ferenc and Tibor Gyimóthy
University of Szeged, Department of Software Engineering
{bakotat|ferenc|gyimi}@inf.u-szeged.hu

Abstract

Although source code cloning (copy&paste programming) represents a significant threat to the maintainability of a software system, problems usually start to arise only when the system evolves. Most of the related research papers tackle the question of finding code clones in one particular version of the software only, leaving the dynamic behavior of the clones out of consideration. Eliminating these clones in large software systems often seems absolutely hopeless, as there might exist several thousands of them. Alternatively, tracking the evolution of individual clones can be used to identify those occurrences that could really cause problems in the future versions.

In this paper we present an approach for mapping clones from one particular version of the software to another one, based on a similarity measure. This mapping is used to define conditions under which clones become suspicious (or "smelly") compared to their other occurrences. Accordingly, these conditions introduce the notion of dynamic ***clone smells****. The usefulness of these smells is validated on the Mozilla Firefox internet browser, where the approach was able to find specific bugs that resulted from neglecting earlier copy&paste activities.*

1. Introduction

In industrial environments, where the developers are under pressure of deadlines, it is a common practise to reuse source code by simply copying parts of it using the clipboard. Although, this approach might decrease software development time, the price in long-term must be paid back in form of increased maintainability efforts. One of the primary concerns is that if the original code segment needs to be corrected, all the replicated parts need to be checked and changed accordingly as well. By inadvertently neglecting to change the related duplications the programmers may leave bugs and introduce inconsistencies to the code.

Finding *code clones* (source code duplications) in large-scale software systems has become an extensive research topic lately. The diverse detection algorithms range from lexical (token-based) [10, 6, 12], through AST-based [4, 26, 17] to metric-based [18, 16, 21, 19] approaches. All these methods operate on one particular version of the software and as a result they provide a detailed list of copied code segments, containing eventually several thousand items. Fowler [22] argues that code duplications are the most important ones among *bad smells* (a particular part of the source code that reflects some kind of design or implementation related flaw) and they should be eliminated aggressively by the programmers. Their speciality lies in that contrary to the other bad smells, the elimination of code duplications is in most of the cases easier (even automatic refactoring techniques exist [3, 2, 14, 15]) and their impact to the code quality is more evident. Some of the latest research results suggest that aggressive elimination of duplicated code might not be the best strategy as it requires a great amount of human resources and there is no guarantee that the costs will pay back in the sense of maintainability improvement. Indeed, Kim et al. [13] found that immediate elimination of volatile clones might not be cost-effective. Additionally, rarely changing duplications may, again, not be worth of immediate elimination as there are usually many of them and they might start causing problems only when they evolve (which happens rarely). Despite this obvious connection of code evolution and cloning, only few of the research papers deal with the dynamic behavior of the duplications through subsequential versions of the system.

In this paper we tackle the question of how individual copied code fragments change through the versions of the system. As the first step, it is necessary to define an *evolution mapping* between two particular code fragments from different versions. Specifically, a clone from a particular version is mapped to a clone in another version, if the subsequent one has evolved from the previous one. This correspondence mapping between the clones is trivial in some special cases. For example, *named entities*[1] are entirely de-

[1] Named entities are syntactic units in the source code that have a unique name which entirely determines their identity. These are the classes, functions, variables, etc.

termined by their unique names[2], therefore the evolution mapping exists between two such entities if their unique names match (the contrary need not be true). For handling the rest of the cases (when there is no such unique name, or the unique names of the considered entities do not match) we introduce a *similarity measure* to determine which code segments are mapped to which other parts.

Using the above mapping, we introduce the notion of *clone smells* which, similarly to the bad smells, refer to a particular code portion that became risky as the result of the latest development activities. The definitions of these smells take into account the dynamic behavior of the clones (e.g. how did they change as the system evolved). The advantage of this approach is that instead of focusing on a list of several thousand copied code segments and eliminating them (even those which will probably never be modified again), the developers can concentrate on those ones which really could contain some flaws as a result of being copied. We present definitions of four different clone smells that can help identifying the risky code parts and can be used even for finding specific copy&paste related bugs.

We validated our results on the Mozilla Firefox [24] internet browser by taking 12 sequential head revisions equally distributed from year 2006. The validation resulted in a list of 60 particular code parts that should be placed under revision. By manual evaluation we found that the obtained results contain 7 such code segments that directly decrease maintainability of the system. In 5 further cases, bugs also might have been introduced. In 2 more cases real code related flaws were found which was also confirmed by the bug tracking system of the underlying software (the found bugs have already been fixed in the mean time). It also turned out that if clone smell detection was used continuously during the underlying period of time at least 4 such segments that were later modified in connection with bugfixes could have been identified.

We will proceed as follows. In the next section we will explain some terms and notions used in the paper. Afterwards, in Section 3 we will discuss some works similar to ours. In Section 4, we will present the details of our approach for creating the evolution mapping between versions. Next, in Section 5 we will deal with the definition and meaning of the clone smells. In Section 6, we will validate the usefulness of the dynamic clone smells on the Mozilla Firefox internet browser. We will close our paper with conclusions and directions for future work in Section 7.

2. Notions

Cloning defines an equivalence relation on the set of copied code segments. Two code segments are in relation if they are considered to be copy of each other (with respect to the underlying clone detection approach). We use the notion of *clone classes* (or *clone groups*) for the classes of the relation, and the members of the classes will be referred to as *clone instances*. By the nature of the relation, each clone class contains at least two clone instances.

[2]The unique name of e.g. a class is its full qualified name together with its path and file name. In case of functions their signature is also added.

3. Related Work

There are several works dealing with diverse approaches for detecting code clones. Starting from the algorithms which are based on lexical comparison of the source lines or tokens [6, 10, 12], through the metric-based approaches [16, 18, 19, 21] which use metrical values of the code parts in order to identify similar fragments to the more sophisticated AST-based approaches [4, 17, 26] which require full syntactic analysis of the source code before the clone detection can take place.

There are much less works dealing with the dynamic behavior of the clones through subsequential versions of the system. One of these works is presented by Antoniol et al. [1] in which they used time-series techniques to model the change of the average number of clones per function in the system. For building such a model it is not required to map the clones from one version to the clones of the other version, just the number of clones in each function is required to be computed. Metrical values of the function were used to identify them across the versions. The model was managed to predict the amount of cloning in the system with an average error of 3.81 %.

Merlo et al. [23] extended the concept of similarity of code fragments to quantify similarities at the release/system level. Similarities were captured by four software metrics representative of the commonalities and differences within and among software artifacts. In that way they were able to model the change of the similarity of the code between the versions of the system without even performing clone-detection directly.

Kim et al. [13] proposed a similar approach to ours. They defined the *Cloning Relationship* between two clone classes based on the lexical similarity of their representatives. In this way a directed acyclic graph was obtained (the nodes are the clone classes and the edges are represented by the evolution pattern relationship). *Clone Genealogy* is a connected component of the above graph where every clone group is connected by at least one evolution pattern. Clone genealogy was used to perform a study on two small-sized Java systems from the aspect of the cloning habit of the developers.

Johnson [11] investigated the clones in two versions of GCC using a text-based matching approach similar to *CCFinder*. In this case, clone detection was used not just to identify duplicates, but also to track changes (e.g. renaming

files) between the two underlying versions of GCC.

Duala-Ekoko [5] et al. proposed a technique for tracking clones in evolving software. They proposed the notion of abstract Clone Region Descriptor (*CRD*) which describes the clone instances within methods in a way that is independent from the exact text or its location in the code. In that way, their *CloneTracker* system is able to keep track of clones even if the code evolves. The attributes used for constructing the *CRD* are similar to those which we used for defining the similarity measure in this paper.

4. Our Approach

Our approach for finding clone smells consists of three steps:

1. Identifying code clones in all the available versions of the software.
2. Computing clone instance evolution for all the extracted clone instances.
3. Identifying clone smells based on the clone instance mappings.

4.1. Clone Detection

Clone detection is preceded by the syntactic and semantic analysis of the source code performed by the *Columbus* reverse engineering environment [8, 9].

For the identification of duplicated codes, we apply the AST-based approach presented by Koschke et al. [17] implemented on the *Columbus schema* [7] instance (which is basically an AST – Abstract Syntax Tree decorated with semantic edges). This approach finds duplicated codes that form syntactic units (e.g. classes, functions, blocks, iterations) with linear time and space complexity. The similarity of the clone instances in case of this approach is defined by the serialization method of the AST. In this context two code parts are similar (and therefore fall into the same clone class) if they consist of the same AST node types (represented by the schema) in the same order. The clone extractor tool reports the list of clone classes and clone instances with their precise location in the source code. Additionally, every clone class has a special node type (*head*) that corresponds to the type of the syntactic unit that is represented by the instances (e.g. function node - if the whole function is a clone instance).

In the following we identify the evolution mappings between clone instances across the versions of the software.

4.2. The Evolution Mapping

The *evolution mapping* is in our context a partial injective mapping of the clone instances of version v_1 to a version v_2 of the subject system. The intuitive meaning is that the images of mapping have evolved from the domain instances. The mapping is considered to be *partial*, as there might be such clone instances that have vanished in the subsequent version. *Injectivity* means that every clone instance from the newer version has evolved out of at most one earlier instance. There are some important properties of these mappings which play a crucial role in the further considerations. Firstly, these mappings are *invertible* (the inverse is also a partial injective mapping). Additionally, the composition of two such mappings is also a partial injective mapping. Lastly, the following requirement is expected to be fulfilled based on the nature of the evolution mappings: let $f : P \to Q, g : Q \to R$ and $h : P \to R$ be three evolution mappings. We know that $f \circ g$ is also a partial injective mapping, but it is also required that $f \circ g = h$. A very simple and natural meaning stands in the background: if a clone instance C_2 has evolved from the clone instance C_1 and C_3 has evolved from C_2 than we expect C_3 to have evolved from C_1. We will refer to this requirement as *Composite Mapping Requirement* (CMR). (In practice h is less reliable than $f \circ g$ as the previous one covers a longer period of time in one step and there might be such evolution patterns that can be discovered only by taking smaller steps in time.)

The idea presented in this section for creating an evolution mapping makes an attempt to simulate human thinking. When trying to assess the question of how humans correspond code parts that have evolved from each other, we concluded that there are several features and constraints which should be taken into account. Starting from the constraints, there is one obvious rule defined in order to eliminate the impossible mappings:

C_1 : The *head* of the clone classes (represented by the two instances) must be the same.

This constraint prevents the mapping of different types of clone instances to each other. For example, it is intuitively clear that a *function node* should not be considered to have evolved from a *class node*. By applying this "cutting" rule, a reasonable amount of computation can be saved, as its verification is simple and fast. For those clone instance pairs that satisfy the rule above, an evaluation of a similarity measure is needed, which is more expensive and which is aggregated from the following features:

F_1 : Name of the file containing the clone instance.

F_2 : Position of the clone instance inside the clone class.

F_3 : The unique name of the *head node* - if the unique name exists (just for named entities).

F_4 : Otherwise, the unique name of the first *named* ancestor in the AST.

F_5 : The relative position of the code segment inside its first named ancestor.

F_6 : Lexical structure of the clone instance.

The intuitive meaning of the features is the following. If two pieces of code are considered and one is trying to determine whether the second one has been derived from the first one, the first thing is to check how similar the two pieces of code are (F_6). If their similarity reaches a subjective level of acceptance, the second question is, whether their relative positions differ a lot, i.e. are the two instances near each other with respect to the containing entity (e.g. function or class) (F_5). If the difference is large they are less likely to correspond to each other. Afterwards, one would check whether the names of these two containing entities are the same or at least are similar to each other (F_4). If there are more clone instances falling into the same code part (function, class, etc.) one would make sure not to interchange the corresponding parts by accident (the evolution mapping should be order preserving) (F_2). If there are multiple occurrences of the subsequent clone, which are contained in different files, one would much likely choose the one whose filename is the most similar to the original one (F_1). In the case of named entities the situation is much simpler: one would just compare the unique names of the two instances (F_3).

It is important to see that the above concepts are fuzzy-like requirements: violating any of them does not cause rejection of the considered mapping automatically, just that the chance of such a mapping would be smaller. For example, if the considered instances are in different files, they could still be in evolution relationship with each other (unless there is another instance with some characteristics which is located in the same file). It is also important to note that the above stated conditions strongly rely on the usage of a syntax-based clone detector. Furthermore, the listed features are not equally important. For example, the lowest priority feature is F_1 whose significance should come out only if no other feature can distinguish between the possible mappings.

Although, the presented evolution mapping is not specific to duplicated code (can be applied to any two fragments of code), it cannot be used independently itself for tracking code evolution, because the technique requires a list of candidates which might have evolved from each other. Otherwise, every code segment of one version should be compared to every segment of the other version, which is computationally unacceptable. The list of candidates is provided by the clone detector.

In the following we will formalize the above stated concepts. For all clone instance pairs that satisfy the cutting rule C_1, the values of the listed features are computed and the results are composed into one similarity measure that reflects in some sense all the features at once.

Each feature F_i contributes to the overall similarity by a given predefined weight α_i. Let C_i and C_j be two particular clone instances, and let $Sim_k\left(C_i, C_j\right)$ be the similarity value of the F_k features of these instances. Furthermore, let

$$Sim\left(C_i, C_j\right) = \sum_{k=1}^{6} \alpha_k Sim_k\left(C_i, C_j\right),$$

denote the overall composed similarity of the features for the two considered instances.

In our context the $Sim_k\left(C_i, C_j\right) = 0$ condition reflects the exact match of the F_k features. As the matching gets worse, the value of the function increases. By considering C_i as a clone instance of the version v_s of the subject software system, and iterating C_j through the instances of version v_t, we define the evolution mapping of C_i in the following manner:

$$\begin{aligned} C_i^t = \quad & \{C_j \in CI_t : min_{C_k \in CI_t} Sim\left(C_i, C_k\right) = \\ & = Sim\left(C_i, C_j\right) \wedge Sim\left(C_i, C_j\right) < \beta\}, \end{aligned}$$

where CI_t stands for the set of all clone instances in version v_t. The formula states that C_i^t is the set of all clone instances in version v_t that are the most similar to the clone instance C_i in version v_s and the similarity value does not exceed a given threshold value β. We assume that this set has at most one element (otherwise we choose among the elements randomly), and we also identify the unique element of it by the whole set itself. In our implementation a *greedy* approach ensures maintaining injectivity of the mapping (if it is violated we take the edge which lower similarity). Figure 1 points out the importance of the threshold value: if we assume that the weights α_i correctly determine the evolution mapping, then, in the first diagram the similarity measure between the instance from $V1$ and $V3$ is high (they are dissimilar) and the realization of this mapping would contradict to the composite mapping requirement. In the case of the second diagram, the edge between the instances of V_1 and V_2 should be deleted. Of course, the case when the α_i weights are bad still remains; the example just shows us that even in case of proper α_i values, the cutting condition cannot be neglected as it is in some sense orthogonal to the weights (it cannot be expressed by them).

There are two questions arising from the above definition of evolution mapping. (1) How can one define the similarity measures for the separate features? (2) How to determine the weights and the cutting threshold value?

As the features F_1, F_3, F_4 and F_6 operate on lexical (string) values, we employed a modified *Levenshtein distance* [20] (also known as *edit distance*) in order to measure their similarities. In our case the distance of two strings is their edit distance divided by the length of the longer one. In this way we obtain a similarity value for each of the above features lying between 0 and 1. By experimentation we found that in the case of the F_6 feature it is not enough to compare the AST node types, but it is much better to use

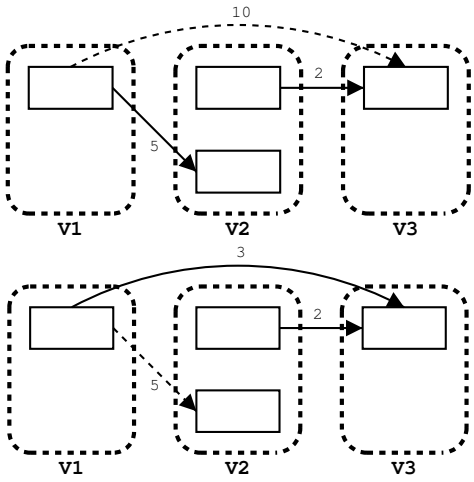

Figure 1: An example of evolution mapping across three versions of a system

the original source code. The reason for this is that the node types of the AST tree represent only a high-level abstraction of the source code: variable names, function names and some other lexical types of information are missing. Hence, if there are two or more instances of the same clone class near each other, they cannot be told apart from each other, even though they might use absolutely different variable names, or they might call different functions. To come around this issue, we utilized the *Columbus Source Generator Library* which is also part of the *Columbus Framework* to generate the source codes for the underlying parts, and the resulting strings were compared by means of textual similarity. With this method the generated code parts were formatted in the same manner, which would not be the general case if the original code fragments were taken.

In case of F_2 and F_5 a different approach is needed, as they represent numerical information. The F_2 feature is responsible for the correct order of the clone instances. Let's suppose that there are two instances of the same class inside the same function body near each other, which are also textually very similar. Our clone detector is implemented in such a way that the clone instances which are located inside one common syntactic unit (e.g. class, function) are sent to the output in the order of their appearance. By relying on this property of the clone detector, the order of appearance of the instances inside their class should also be taken into consideration as the rest of the features cannot distinguish the above code fragments.

More specifically, if C_i and C_j are two instances, and $O(C_k)$ is the order of appearance of the C_k instance inside its class, then $Sim_2(C_i, C_j) = |O(C_i) - O(C_j)|$. If the order of C_i and C_j are the same, the feature F_2 does not contribute to the overall similarity measure. The situation is a little bit more complicated in case of F_5. This feature is responsible for measuring the disalignment of the instances with respect to their parent. Let $l_1(C_i)$ be the number of AST nodes from the beginning of the parent node to the beginning of the instance, let $l_2(C_i)$ be the length of the instance (measured by the number of AST nodes), and let $l_3(C_i)$ be the number of nodes from the end of the instance to the end of the parent node. Using the notions above, let

$$Sim_5^2(C_i, C_j) = \left(\frac{l_1(C_i)}{l_1(C_j)} - L_{ij}\right)^2 + \left(\frac{l_3(C_i)}{l_3(C_j)} - L_{ij}\right)^2$$

where

$$L_{ij} = \frac{l_1(C_i) + l_2(C_i) + l_3(C_i)}{l_1(C_j) + l_2(C_j) + l_3(C_j)}.$$

According to the above definitions, two code segments are similar if the differences in the amount of AST nodes before and after the segments do not differ much with respect to the overall size of the AST starting from the parent nodes. For example, if the code segments are located inside a function node, the value of Sim_5 will remain small even if the body of the function is stretched linearly (approximately the same percentage of new AST nodes are added before and after the considered segment). In this way we get a "stretching-invariant" disalignment similarity measure.

At last, we have arrived to the question of weights and cutting values. Fortunately, the weights are meaningful, therefore many of them can be set by relying on human experience. Even if the optimal values are not easy to determine, the mutual priorities of the features offer themselves. For example, the similarity of the code fragments (F_6) is obviously more important than their relative location inside their parent node (F_5). Provided that the two segments differ a lot on lexical level, we would take the one which is more similar, instead of the one whose position is more adequate. The reason for this is that programmers are much likely to alter the position of the segments (e.g. by inserting/deleting code lines) than to change the lexical structure of it (e.g. by renaming variables or calling other functions). Similarly, the name of the file is the least important: it should be considered only if nothing else remains.

Assigning initial weights with consideration of the empirically justified priorities leads us to a starting point of an iterative approach. By fixing the weights it is easy to evaluate the goodness of the resulting mapping without any manual validation. It is enough to measure how the composite mapping requirement is fulfilled by the resulting correspondence. By taking three consecutive versions, creating the evolution mappings between any two pairs (using the fixed α_i weights and the β threshold) and by measuring the amount of CMR violations, gives an approximate measure regarding the goodness of the weights. The amount of CMR violations can be expressed by means of the following three constraints which are equivalent to the original definition of CMR (if these are satisfied there is no CMR violation in the model):

1. for any versions v_t and v_u, where $v_u < v_t < v_s$, if $C_j \in CI_{v_t}$ and $C_k \in CI_{v_u}$ such that $C_j^{v_s} = C_i$, $C_k^{v_s} = C_i$ and there is no $C_l \in CI_{v_u}$ such that $C_l^{v_t} = C_j$ then $C_k^{v_t} = C_j$.
2. for any versions v_t and v_u, where $v_u < v_t < v_s$, if $C_j \in CI_{v_t}$ and $C_k \in CI_{v_u}$ such that $C_k^{v_t} = C_j$ and $C_j^{v_s} = C_i$ then $C_k^{v_s} = C_i$.
3. for any versions v_t and v_u, where $v_u < v_t < v_s$, if $C_j \in CI_{v_t}$ and $C_k \in CI_{v_u}$ such that $C_k^{v_s} = C_i$ and $C_k^{v_t} = C_j$ then $C_j^{v_s} = C_i$.

As the goodness of the model can be easily evaluated, the need for applying an optimization algorithm arises naturally. We decided to use the *gradient method (steepest descent)* [25] in order to optimize the values of the weights. Being able to find local extreme values only it is recommended to set the initial values accordingly. The optimization was performed by using 12 versions of the *Mozilla Firefox* internet browser (equally distributed in time from year 2006) by taking all the 10 consecutive triplets and evaluating the amount of violations of the above constraints. The cost-function that we have chosen to minimize was the following:

$$\mathcal{C}(\vec{\alpha}) = \frac{CMR(\vec{\alpha})}{E(\vec{\alpha})},$$

where $CMR(\vec{\alpha})$ stands for the goodness of the mapping, and $E(\vec{\alpha})$ is the overall number of edges represented by the evolution mappings. The number of edges should be maximized at same time as it would be easy to set up a model which fully satisfies the CMR: let β be zero (the evolution mappings are empty). It is important to notice, that the degree of freedom of the model is one, which means that by multiplying all the α_i and β values with the same positive constant would not change the model itself (the resulting mappings will be the same). Therefore we may assume that the optimal choice for β is equal to 1, and proceed with the optimization of the rest of the variables with respect to this constraint. Table 4.2 shows the initial values of the weights and the values obtained during the optimization. It can be seen that basically all the initial values were overestimated (with consideration to the fixed β value). Furthermore, it turns out that feature F_2 contributes to the differentiation of the code segments with a higher factor than F_5 (contrary to our initial assumption). The rest of the weights basically reflect our expectations: textual similarity (F_6) is the most important (90.6%) and for the named entities the unique name (F_3) affects the decision in 60%. File name (F_1) is the least distinguishing factor with its 7.9% of contribution.

The model which operates with the optimized weights, violates CMR in 25 cases (this is a summarized number for all the 10 triplets), and the number of edges is 178,026.

Now, we are ready to define the dynamic clone smells based on the evolution mapping defined in this section.

Weights	Initial	Optimized
α_1	0.1	0.0793
α_2	0.3	0.2911
α_3	0.8	0.6002
α_4	0.2	0.0943
α_5	0.5	0.2312
α_6	1.0	0.9066

Table 1: Initial and optimized weights of the model

5. Dynamic Clone Smells

There are several scenarios when manipulating clone instances may cause problems either in the subsequent version or later in the future. In the following, we will summarize some of these cases and give a short description of their meaning and consequences.

Vanished clone instance (VCI)

Definition: $C_i \in CI_{v_s}$ is a vanished clone instance if $C_i^{v_{s+1}} = \emptyset$, i.e. C_i is not mapped onto any instance of the subsequent version.

There can be several reasons for this smell:

- The corresponding code part was removed in the subsequent version. If this is the case, there is nothing to do, the maintainability of the system improved as the redundancy has been eliminated.
- The instance has changed so much that the evolution mapping was not able to discover the relationship between the two code segments. By optimizing the set of weights the amount of these false positives can be reduced.
- The instance has changed in such a manner that it is not a clone of other code segments any more. For the clone detector this part of the code becomes invisible, so there is nothing to map on. This is the critical case, as the developer might have forgotten for the other instances which should have been modified accordingly by chance. Even if the modification was not critical (e.g. the true branch of an *if* statement was placed into a block) the modifications should have been repeated for all the instances in the clone class, otherwise the clone detectors might not be able to find these parts any more. By loosing the connection between the copied code segments the maintainability is reduced significantly as the subject code portions might still provide the same functionality but they become undiscoverable.

Occurring clone instance (OCI)

Definition: $C_i \in CI_{v_s}$ is an occurring clone instance if $C_i^{v_{s-1}} = \emptyset$, i.e. C_i is not an image of any instance from the previous version.

Possible meanings of the smell:

- A new instance of duplicated code was found. Developers might need to consider refactorings in order to eliminate the newly encountered redundancy to maintain system quality.
- Earlier modifications (bug-fixes, improvements) of other code portions (that were originally clones of the considered one) were applied to this code segment as well, making it again a member of the original clone class. Elimination of these instances is highly recommended as they have already caused problems and will probably cause problems in future again.

Moving clone instance (MCI)

Definition: $C_i, C_j \in CI_{v_s}$ being representatives of the same clone class, are moving clone instances if $C_i^{v_{s+1}}$ and $C_j^{v_{s+1}}$ are not in the same class any more, i.e. one of them has moved to another class (it is not always clear which one, as the members of the original class might have changed as well). In general, this case can be considered as if the original clone class was broken apart, therefore the smell is reported for the class itself.

Possible meaning of the smell:

- Developers might have modified a code which became a clone instance of another class. This can happen either accidentally, in which case the programmer might not know that a similar code already exists. Similarly, the developer might not be aware whether the rest of the occurrences should be replaced or not. On the other hand if the new code of the instance is copied intentionally, only the second case of the above notions holds. In any of the cases checking the smell could help maintaining code quality.

Migrating clone instance (MGCI)

Definition: $C_i, C_j \in CI_{v_s}$ are representatives of the same clone class, $C_i^{v_t}, C_j^{v_t} \in CI_{v_j}$ are in different clone classes for some $t > s$ and $C_i^{v_u}, C_j^{v_u}$ are again in the same clone class for $u > s$. The definition considers those instances that were "together" at the beginning, they separated for a while, and at last they came together again. The smell is reported for pairs of instances.

Possible meaning of the smell:

- An instance of a clone class was modified earlier, so it got out from its class. The developers might have forgotten to modify the other instances of the same class. Later on, when this mistake was discovered (possibly by encountering a bug) an other instance of the original class was modified in the same manner (bug-fix). At this moment a clone smell is reported as it became highly probable that modifications were not intentionally neglected in the first round. If this is the case, the remaining instances of the original class should be considered as well. This smell is more restrictive than the others as the underlying instance might either vanish (VCI) – it will be undiscovered for a while – or just move to another class (MCI). The instance which is changed later just moves to a new class (MCI) while the original one either appears (OCI) or not.

The disadvantage of this smell is that when it is reported, the flaw might have already been fixed (at least if there are no remaining instances in the original class that have not migrated yet). On the other hand, reporting an MGCI smell is always preceded by either a VCI or an MCI smell. These two smells should be considered by the developers in order to shorten the migration period of the remaining instances. Even if the original class does not contain any other instances (the migration of instances has finished), we can still use the smell to approximate how many flaws could have been discovered during a period of time if the continuously reported VCI and MCI smells were evaluated.

In the following we will prove the justification of the smells.

6. Validation

We validated the usefulness of the dynamic clone smells on the Mozilla Firefox internet browser. The analysis and clone detection of 12 consecutive versions of the head revision (which were taken equally distributed from year 2006) was performed by the *Columbus framework*. The overall time for this preparation step took approximately 2 hours on an *IBM BladeCenter LS20* machine equipped with 10 modules, each of them operating with two AMD Dual Core Opteron 275 processors, running on 2,2 GHz containing 4 GB of memory each. Table 6 shows in which range do the basic metrical values of the underlying software system change.

Metrics	Value range
lLOC(logical Lines Of Code)	1,245,450 - 1,356,623
NCL (Number Of Classes)	3,770 - 4,079
CI (Clone Instances)	5,774 - 6,139
CCL (Clone Classes)	2,370 - 2,533
CC (Clone Coverage)	6 % - 6 %

Table 2: Basic metrical values of Mozilla Firefox

In the next step the construction of the evolution mapping was performed based on the optimized weights computed as presented previously. Evolution mapping was constructed between any two considered versions, as for the MGCI smell would not have been enough to take the consecutive systems only because of the possible vanishing and reappearing instances. This step was not parallelized and therefore it took approximately 3 hours to finish on one processor core. The amount of required computation was notable because every clone instance from one version had to

be compared to every other from the second version and this had to be repeated for any two versions.

After extracting the clone smells we analyzed them manually in order to check their effect on software maintenance. We also tried to find the root causes of the reported smells. It was expected to get a relatively short list of smells that could provide valuable clues regarding maintaining the actual quality of the considered software system. Table 6 summarizes the clone smells found in Mozilla Firefox. The columns of the table refer to the clone smells, while the rows describe the reasons that might have caused them.

Type	**VCI**	**OCI**	**MCI**	**MGCI**
False positives				
Irrelevant clones	3	3	3	3
Compilation problem	2	2	-	2
Bad evolution mapping	4	4	1	1
True positives				
Non-semantic modification (loosing cloning relationship)	6	-	1	1
Non-semantic modification (Reestablishing earlier cloning relationship)	-	2	-	-
Instance Deleted	3	-	-	-
Replicated codes not modified (possibility of remaining bugs)	2	-	3	-
Real bugs introduced (Already fixed since then)	2	-	-	4
Resulted from bugfixes	-	8	-	-
Overall amount	22	19	8	11

Table 3: Amount of clone smells in Mozilla Firefox

The clone smell extractor was able to identify 22 vanishing instances out of which 2 resulted from bad compilation (some of the source files of the Mozilla Firefox were not compiled due to syntax errors). These false positives cannot be backed out as there is no way to verify whether some particular instances are missing because of compilation errors or because they were just eliminated. All the clone smells that resulted from compilation failures were originated from one particular source file that failed to build. Four of the reported VCI smells were caused by the mistake of the evolution mapping. Although the mapping is highly reliable because of the earlier optimization, the outcome is very sensitive to the imprecision of the mapping itself. Improving the similarity measure could help reducing the amount of smells falling into this category. The clone detector also made a mistake (it found a false positive). It identified one clone class (containing three instances) which it is not real code duplication (the similarity existed on the AST level only). These are referred to as irrelevant clones in the table.

Six VCI smells were reported because of an ineffective change to the code (just structural, not affecting the semantics) which will make the corresponding code parts invisible for the clone detectors in the future. Applying the same changes to related code parts as well would remove this inconsistency.

One MCI smell was reported because of the same reason, but in this case the instances did not vanish, just the original class was broken apart as only some of its instances were modified. The same class was also reported by the MGCI smell meaning that the same modifications were indeed applied to the rest of the instances later as well.

Pure structural modifications may also cause new clone instances. This is reported by the OCI smell in the second row of the true positives. In this case the two code segments were readjusted. Normally, it would be reported as MGCI smell as well, but the instances were broken apart before our analysis started.

In further three cases VCI smells were reported because the underlying code part was eliminated which reduces the redundancy in the code and therefore improves quality. The above categories do not reflect directly any flaw in the code but they point out those part which could cause a drop in quality.

The following cases refer to more serious problems which may have direct impact on code quality. The VCI smell reported two cases when the instances simply vanished after applying a not pure syntactical change which might have altered the overall functionality of the underlying code part. The copied code segment was not modified accordingly so it should be checked at least to see if it had been forgotten or intentionally neglected. In other two cases it is obviously neglected by accident as the same changes were applied weeks later in connection with a bug-fix (the information was checked in the *Bugzilla* bug tracking system of Mozilla). Figure 2 shows the exact original and modified code pieces which could have been used to identify this bug earlier.

Modifying only some of the instances of a clone class caused three MCI warnings, meaning that the corresponding instances did not vanish but their class was split into more pieces. Four MGCI smells were found to be in connection with real bugs. By the definition of MGCI, the smell is reported only if at least one of the forgotten instances has been adapted to the already changed one. In this case it means that based on the VCI and MCI smells four code related flaws could have been backed out during the underlying period of time if clone smells were computed continuously.

It can be seen in the last row of the true positive section that eight instances appeared during this period thanks to bug-fixes. All these instances were in pairs and had been broken apart before the start of our analysis. If clone smell detection had started before they were separated, all of them would have been reported much earlier either by a VCI or by an MCI smell.

```
328 NS_IMETHODIMP_(nsISVGGlyphFragmentLeaf *)
329 nsSVGTSpanFrame::GetNextGlyphFragment()
330 {
331   nsIFrame* sibling = mNextSibling;
332   while (sibling) {
333     nsISVGGlyphFragmentNode *node = nsnull;
334     sibling->QueryInterface(NS_GET_IID(nsISVGGlyphFragmentNode),
          (void**)&node);
335     if (node)
336       return node->GetFirstGlyphFragment();
337     sibling = sibling->GetNextSibling();
338   }
339
340   // no more siblings. go back up the tree.
341
342   NS_ASSERTION(mParent, "null parent");
343   nsISVGGlyphFragmentNode *node = nsnull;
344   mParent->QueryInterface(NS_GET_IID(nsISVGGlyphFragmentNode),
          (void**)&node);
345   return node ? node->GetNextGlyphFragment() : nsnull;
346 }
```

nsSVGTSpanFrame.cpp (June 6, 2006)

```
850 NS_IMETHODIMP_(nsISVGGlyphFragmentLeaf *)}
851 nsSVGGlyphFrame::GetNextGlyphFragment()
852 {
853   nsIFrame* sibling = mNextSibling;
854   while (sibling) {
855     nsISVGGlyphFragmentNode *node = nsnull;
856     sibling->QueryInterface(NS_GET_IID(nsISVGGlyphFragmentNode),
          (void**)&node);
857     if (node)
858       return node->GetFirstGlyphFragment();
859     sibling = sibling->GetNextSibling();
860   }
861
862   // no more siblings. go back up the tree.
863
864   NS_ASSERTION(mParent, "null parent");
865   nsISVGGlyphFragmentNode *node = nsnull;
866   mParent->QueryInterface(NS_GET_IID(nsISVGGlyphFragmentNode),
          (void**)&node);
867   return node ? node->GetNextGlyphFragment() : nsnull;
868 }
```

nsSVGGlyphFrame.cpp (June 6, 2006)

```
207 NS_IMETHODIMP_(nsISVGGlyphFragmentLeaf *)
208 nsSVGTSpanFrame::GetNextGlyphFragment()
209 {
210   nsIFrame* sibling = mNextSibling;
211   while (sibling) {
212     nsISVGGlyphFragmentNode *node = nsnull;
213     CallQueryInterface(sibling, &node);}
214     if (node)
215       return node->GetFirstGlyphFragment();
216     sibling = sibling->GetNextSibling();
217   }
218
219   // no more siblings. go back up the tree.
220
221   NS_ASSERTION(mParent, "null parent");
222   nsISVGGlyphFragmentNode *node = nsnull;
223   CallQueryInterface(mParent, &node);
224   return node ? node->GetNextGlyphFragment() : nsnull;
225 }
```

nsSVGTSpanFrame.cpp (July 2, 2006)

```
832 NS_IMETHODIMP_(nsISVGGlyphFragmentLeaf *)
833 nsSVGGlyphFrame::GetNextGlyphFragment()
834 {
835   nsIFrame* sibling = mNextSibling;
836   while (sibling) {
837     nsISVGGlyphFragmentNode *node = nsnull;
838     sibling->QueryInterface(NS_GET_IID(nsISVGGlyphFragmentNode),
          (void**)&node);
839     if (node)
840       return node->GetFirstGlyphFragment();
841     sibling = sibling->GetNextSibling();
842   }
843
844   // no more siblings. go back up the tree.
845
846   NS_ASSERTION(mParent, "null parent");
847   nsISVGGlyphFragmentNode *node = nsnull;
848   mParent->QueryInterface(NS_GET_IID(nsISVGGlyphFragmentNode),
          (void**)&node);
849   return node ? node->GetNextGlyphFragment() : nsnull;
850 }
```

nsSVGGlyphFrame.cpp (July 2, 2006)

Figure 2: Vanishing clone instance indicates a bug which was fixed in August 2, 2006 with the following log message: "Bug 290766 - Use CallQueryInterface in frame code"

7. Conclusion and Future Work

There has been many research papers published regarding diverse clone detection techniques which operate on one particular version of a software system. But code duplications should not only be considered independently from code evolution, because their presence is mostly being emphasized while the code changes. In this paper we made an attempt to connect separate clone instances across several consecutive versions of a system.

We found that an appropriate machine learning algorithm can be trained in order to relate those code segments across the versions which have evolved from each other. By relating clones from different versions we were able to provide a list of exactly defined code segments of a large software system that could contain flaws or might cause maintenance related problems in the future. The final conclusion is that efforts directed towards connecting code evolution and code duplications have their *raison d'être*.

In the future, we intend to improve the proposed similarity measure as its reliability significantly affects the amount of false positives. The presented *gradient descent* optimization method might not be the best choice in this case because the cost-function behaves weirdly on its domain (it is not even continuous, so the derivatives were obtained by rough approximations), therefore it easily gets stacked in local extremals. Thus, other approaches like *simulated annealing* could yield better results. Furthermore, the evolution mapping gives rise to clone evolution based measurements which could also reflect the change of code quality. Experimenting with further smells could also point out code or maintenance related flaws.

Although the implementation is optimized in the sense that the particular attributes are computed in cost-increasing order, and if at any point the threshold value is exceeded the rest of the features will not be considered, the computation effort required is still notable. The reason for that is the MGCI smell which makes the computation be squarely proportional to the number of overall clone instances in all versions. Further optimizations can be performed, but seemingly the above relationship cannot be backed out if adhering to the MGCI smell. For the rest of the smells only the sequential versions had to be considered so the amount of computation is linearly proportional to the number of overall clone instances in all versions.

References

[1] G. Antoniol, G. Casazza, M. D. Penta, and E. Merlo. Modeling clones evolution through time series. In *Proceedings of the 17th International Conference on Software Maintenance (ICSM 2001)*, pages 273–280. IEEE Computer Society, 2001.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99)*, pages 326–336. IEEE Computer Society, 1999.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, pages 98–107. IEEE Computer Society, 2000.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*, pages 368–377. IEEE Computer Society, 1998.

[5] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[6] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM'99)*, pages 109–118. IEEE Computer Society, 1999.

[7] R. Ferenc and Á. Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, Mar. 2002.

[8] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[9] FrontEndART Software Ltd. `http://www.frontendart.com`.

[10] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research (CASCON'93)*, pages 171–183. IBM Press, 1993.

[11] J. H. Johnson. Substring matching for clone detection and change tracking. In H. A. Müller and M. Georges, editors, *Proceedings of the International Conference on Software Maintenance (ICSM '94),* (Victoria, B.C.; Sept. 19-23, 1994), pages 120–126, September 1994.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[13] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Software Engineering Notes*, 30(5):187–196, 2005.

[14] R. Komondoor and S. Horwitz. Effective automatic procedure extraction. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC'03)*, pages 33–43. IEEE Computer Society, 2003.

[15] R. V. Komondoor. *Automated duplicated code detection and procedure extraction*. PhD thesis, 2003. Supervisor-Susan Horwitz.

[16] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. In *Reverse engineering*, pages 77–108. Kluwer Academic Publishers, 1996.

[17] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[18] 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 *Proceedings of the 13th International Conference on Software Maintenance (ICSM'97)*, page 314, Washington, DC, USA, 1997. IEEE Computer Society.

[19] F. Lanubile and T. Mallardo. Finding function clones in web applications. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 379–388. IEEE Computer Society, 2003.

[20] Levenshtein distance. `http://en.wikipedia.org/wiki/Levenshtein_distance`.

[21] G. A. D. Lucca, M. D. Penta, and A. R. Fasolino. An approach to identify duplicated web pages. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment (COMPSAC'02)*, pages 481–486. IEEE Computer Society, 2002.

[22] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[23] E. Merlo, M. Dagenais, P. Bachand, J. S. Sormani, S. Gradara, and G. Antoniol. Investigating large software system evolution: The linux kernel. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment (COMPSAC'02)*, pages 421–426. IEEE Computer Society, 2002.

[24] The Mozilla Firefox Homepage. `http://www.firefox.com`.

[25] J. A. Snyman. *Practical mathematical optimization: an introduction to basic optimization theory and classical and new gradient-based algorithms*. Springer Verlag, 2005.

[26] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.