Stereo: editing clones refactored as code generators

Nic Volanschi
Metaware Technologies & Inria Bordeaux
Email: eugene.volanschi@inria.fr

***Abstract*—Clone detection is a largely mature technology able to detect many code duplications, also called clones, in software systems of practically any size. The classic approaches to clone management are either clone removal, which consists in refactoring clones as an available language abstraction, or clone tracking, using a so-called linked editor, able to propagate changes between clone instances. However, past studies have shown that clone removal is not always feasible due to the limited expressiveness of language abstractions, or not desirable because of the abstraction overhead or the risks inherent to the refactoring. Linked editors, on the other hand, provide costless abstraction at no risk, but have their own issues, such as limited expressiveness, scalability, and controllability. This paper presents a new approach in which clones are safely refactored as code generators, but the unmodified code is presented to the maintainers with the same look-and-feel as in a linked editor. This solution has good expressiveness, scalability, and controllability properties. A prototype such editor is presented along with a first application within an industrial project.**

I. Introduction

Research on software clones over several past decades resulted in readily available tools for clone detection (*e.g.* [3], [20], [6], [12]), able to find pieces of identical or similar code, called *clones*, within software systems written in various programming languages. Clone detectors are nowadays sufficiently scalable to handle software of any practical size, including for instance large inter-projects repositories [26].

Clones reported by clone detectors are usually classified in the following types, according to the variations between instances [19]: Type-1, or exact clones, are identical code fragments modulo whitespace and comments; Type-2, or parametric clones, are identical fragments, modulo whitespace, comments, and substituting some identifiers, literals, types, etc.; and Type-3, are copied fragments with statements added, changed, or removed, in addition to the previous variations. Beyond this classes, the notion of structural clones has been introduced to describe more global similarities, defined hierarchically as recurring configurations of entities containing entities already recognized as clones of each other [15]. For instance, a structural clone is a program family skeleton.

Even though the exact impact of clones on software quality has been a subject of debate (see, *e.g.*, [7]), the clone information reported by clone detectors is generally considered important for maintenance, because in many cases clone instances have to be evolved in consistent ways, as bugs may be introduced otherwise [2].

One possible clone management strategy is to remove clones by refactoring them with abstractions available in the language, such as functions or macros. However, removing clones by program refactoring is sometimes impossible because of limited expressiveness of available language abstractions, or undesirable due to the cost of increased abstraction [9] and the risks when refactoring working code [10].

An alternative clone management strategy proposed in the past has been linked editing [9], [8], [5], [11]. Linked editors record and maintain clone information as some form of metadata associated to software, and propagate evolutions between clone instances, without refactoring the program at all. Linked editing provides thus costless abstractions without refactoring risks, but has not been largely adopted in practice.

In this paper, we spot some important issues of existing linked editors that prevent them from solving the code duplication problem in general. Issues include limited expressiveness, lack of controllability, and very limited scalability. To overcome these limitations of existing linked editors, this paper proposes a new hybrid approach, in which the code is safely refactored using code generators but this refactoring is made essentially transparent to the maintainer by appropriate support in the editor. A prototype editor called Stereo demonstrating the concept is presented along with a first use on a real case.

Our contributions can be summarized as follows:

- we introduce a new hybrid approach for managing clones, which consists in removing clones while still presenting the original code, decorated with clone information, and with linked editing enabled
- we detail an algorithm for modifying a code generator in a very intuitive manner by editing its generated code
- we present a prototype of such editor for C/C++ programs integrated in a state-of-the art IDE, and a first application of it on a case study within an industrial project. However, our approach can be applied to other programming languages, as the textual code generators we use are host-language independent.

Before presenting our solution, starting with Section IV, we first detail the limitations of the available techniques in Section II, and illustrate them on a concrete case study in Section III.

II. Limitations of clone management strategies

The classic strategy to ensure consistent modification of clone instances, when a clone is considered relevant, consists in removing the clone by refactoring it with a program abstraction available in the language, such as a function, a method, a class, or a macro. Many tools and techniques have been described that automatically list potential de-cloning refactorings and sometimes propose them interactively (*e.g.* [4], [21], [1], [22], [23]). However, previous studies have shown that a large proportion of clones cannot be refactored this way [24] due to three main reasons.

Limited expressiveness: Some of the available abstraction mechanisms cannot express naturally certain kinds of variations between clone instances. For instance, Tairas and Grey [4] found that only 19% of the clones reported by the Deckard clone detector on 9 open source systems could be refactored using a tool for clone refactoring called CeDAR. Other studies [25] have shown that even software written by elite developers, such as the standard Java Buffer Library and the C++ Standard Template Library (STL) contains many clones that could not be refactored even with most advanced language mechanisms, respectively Java generics and C++ templates. The main underlying issue is that generic libraries generally involve many clones that cannot be simply parameterized by types or constants.

Abstraction cost: Even when refactoring *is* technically possible with the available language mechanisms, it may not be desirable in many cases, because of the cost of abstraction, in terms of creation, comprehension, use, and evolution [9]

Refactoring risk: In many cases, possible refactorings are not performed simply because of the risks incurred when modifying already tested code, and especially business-critical code [10]. We encounter ourselves this customer-side barrier on virtually any large refactoring project of our company.

To address these limited applicability of clone refactoring, a second, more lightweight, strategy has been proposed, in the form of linked editors [9], [8], [5], [11]. In this strategy, clones are not removed, but only tracked by the development environment, and the linked editor uses this information to alert developers whenever a change concerns cloned code, and/or automatically propagate changes to other instances. If desired, clone instances can be decoupled interactively, leaving the possibility to evolve them independently. Clone information can be either imported from clone detectors or inferred from copy/paste editing actions by the editor itself.

Linked editors solve two main problems of clone removal: they do not incur any abstraction cost, as no new language mechanism is being introduced, and the risk of program refactoring is avoided, as no transformation is performed on the code. However, current linked editors still suffer from a number of important limitations:

Expressiveness: Each tool takes into account some limited kinds of variations between clone instances. For instance, CloneBoard [5] takes into account certain type-3 clones, but not when the added statements are in the middle of one instance. When type-3 clones are expressible, for instance in CodeLink [9], the added statements cannot be conditioned by some clone parameter. Structural clones are not taken into account either.

Scalability: Maintaining links between more than two clone instances is typically not supported (*e.g.* in CloneTracker [8] and CSeR [11]) or not scalable (*e.g* CodeLink involves a differencing algorithm of complexity $O(s^n)$ for n instances of size s).

Controllability: The algorithms used for propagating changes between instances, usually based on differencing, are sometimes complex and opaque to maintainers. According to an evaluation of the own authors of CloneTracker, around 80% of the propagations are done as intended [8]. This means that propagations must be checked by maintainers on each instance. More generally, maintainers do not control the clones metadata: when a propagation fails, they cannot know exactly why, nor optimize the metadata so as to better guide the tool.

As the result of these various limitations, maintainers of cloned code face a dilemma between, one one hand, adopting a refactoring strategy with associated abstraction costs and transformation risks, and on the other hand, using a linked editor with limited expressiveness, scalability, and controlling propagations manually.

The next section illustrates the mentioned limitations of available approaches on a real case study.

III. Case study

Our company is specialized in the modernization of legacy software. This primarily involves the migration of software assets from obsolete platforms (e.g., old mainframes) towards modern platforms (e.g. Unix/Linux). In particular, several migrations we performed on large software assets (typically, 1 to 5 million of lines of code) also involved project-specific clone refactorings.

However, for illustrating the clone management problem and our proposed solution, we take here as a case study a library developed within our company for a migration project, which contained a great proportion of cloned code. The library consists of a main class called Decimal class emulating fixed-point financial computations on a Unix platform, as provided natively on the IBM z/OS mainframe platform. This library was developed for migrating C/C++ projects between the two platforms. The Decimal class is a critical part of the runtime support for migrated code, despite its small size — 2500 lines located in one single file called "decimal.h" —, because of the potential consequences of fixed-point computation errors in financial applications. Before our clone analysis began, a stable production version (named v1.3) was already up and running with the migrated version of a large (4 MLOC), business-critical code belonging to a major french financial customer. Therefore, the Decimal class represented a typical example where the refactoring cannot be done without strong correctness guarantees.

This relatively modest example is appropriate as a first case study for two reasons. Firstly, the library is written in C++, disposing of one of the most flexible templates mechanisms, compared to other widely used languages. Thus, any clones indicating the limited expressiveness of the language would most probably transpose to many other languages as well. Secondly, the library is not large, and has been recently developed from scratch by a small team of developers, so the code is still well understood by its original developers, and it would be possible to refactor some clones if the improvements were obvious. This increases the chances of finding clones that have good reasons to subsist to traditional refactoring methods, and needing a more advanced support.

A. Clone detection

We analyzed version v1.3 of the Decimal class with the Simian clone detector[1] using the default parameters and found

[1] http://www.harukizaemon.com/simian/

23 type-1 clones of at least 6 lines covering 25% of the code.

We first proceeded to a global code review to identify the general causes of this relatively high cloning rate. This global code inspection revealed the fact that the Decimal class has to provide many similar methods, varying in subtle ways. For instance, all conversion operators between decimal and integral types are very similar except for some differences due to the size of the particular integral type (signed or unsigned int, long, long long, etc.). Furthermore, this similarity in the behavior of different methods favoured a programming practice heavily using copy and paste. Such cloning reasons are commonly encountered in the cloning literature.

Note that these reasons explain why the clones were created in the first place, but not why they were subsisted in the code, without being refactored. To understand that, we proceeded to a detailed investigation of each of the clone sets. More precisely, we inspected the reported clone sets together with the developers of the Decimal class. Thus, we filtered out some irrelevant clones (similar fragments that were meant to evolve independently), we grouped some neighboring type-1 clones in bigger type-2 or type-3 clones, fused together two clone sets, and incidentally found a few relevant clones not reported by the tool. As a result, we selected a total of 17 relevant clones of up to 50 lines, as follows: 4 type-1 clones, 8 type-2 clones, and 5 type-3 clones. Each of the 17 clones contained between 2 and 24 instances, for a total of 110 clone instances.

It turns out that among the 17 clone sets, only 3 could have been refactored easily using existing C++ mechanisms, namely constructor delegation (1 case) and templates (2 cases). For 3 other clone sets, refactoring was technically possible but not desired because of the implied risk (1 case) or increased abstraction cost (2 cases). In these 2 latter cases, the clone covered part of a big switch statement, and extracting this code as a method would create an abstraction difficult to explain. The other 11 clone sets have not been refactored using C++ mechanisms, either because the variable part was not a type but an operator such as '+' or '<=' (5 cases), or they conditionally contained statements or declarations depending on a Type-3 clone parameter (5 cases), or they spanned method startup code not extractible as a separate method: local variable declarations and a few statements (1 case). Thus, developers decided that 14 out of the 17 clone sets were justified to subsist in the code, as they corresponded to the typical limitations of the refactoring approach: limited abstraction (11 cases), high abstraction cost (2 cases), and risk of non-trivial refactoring (1 case). Nevertheless, developers agreed that they had to be maintained in a consistent way.

On the other hand, available linked editors did not offer a credible approach to the developers for these 14 clone sets, for the following reasons. The declarations conditionally included depending on a Type-3 clone parameter (5 cases) are not expressible in such editors (expressiveness limitations). Furthermore, for editors admitting Type-3 clones (without relating the variable parts to a clone parameter), there is a controllability issue: if a new declaration or statement is added next to the variable part in one clone instance, will it be kept private to this instance, or propagated to other instances containing the same variable part, or to all instances? Depending on the intent of the programmer, either strategy inferred by the linked editor may be inappropriate. Finally, for clone sets featuring 12 to 24 instances (5 cases), it would be tedious and error-prone to verify the propagated changes on each instance at every maintenance.

Thus, this case study concretely illustrates the dilemma, mentioned in the previous section, the programmers are facing, due to the limitations of classic approaches to clone management. The following section explores a new direction for widening this limited choice between code refactoring and linked editing with a hybrid approach.

IV. Solution overview

It has been shown [13], [14] that it is possible to *safely* refactor a large variety of clones using code generators, if the refactored program is such that executing the code generators produces back the original, cloned program; this property has been called *iso-generation*.

As a complement to such a refactoring, our solution makes it possible to present the original code to the maintainers by using an appropriate editor, with a look-and-feel similar to a linked editor. More precisely: special support is added in the editor to mark generated code (*i.e.* the clone instances) using a specific color, and to allow maintainers modifying a code generator by *editing the generated code* of any of its instances; upon saving the instance, the updated code generators are re-executed, with the effect of propagating changes in all instances.

We implemented this special editor support in a prototype editor for maintaining C/C++ programs. The editor is written as an Eclipse plugin called Stereo, extending the CDT plugin — the standard Eclipse plugin for editing C/C++ code.

This approach offers much the same advantages as linked editors over classical clone removal:

Refactoring risk: Due to the iso-generation property, the refactored code can be proved to be equivalent to the original code.

Abstraction cost: As the original program is displayed, there is no comprehension overhead; on the contrary, clones instances and their variations are clearly marked, thus helping the understanding of duplicated code. Some maintenance of the generators can be done by editing the generated code, so maintainers are shielded from the extra complexity of the generators during many maintenance scenarios.

Moreover, our approach improves over linked editors along the following dimensions:

Expressiveness: Code generators can easily express clones of types 1, 2, and 3, as well as many kinds of structural clones. For instance, repetition of a type-2 clone over a range of parameter values can be easily expressed by a loop in the code generator.

Scalability: Any number of instances of a clone can be refactored as a single code generator. The overhead of executing all the code generators is linear in the size of the program. This cost is paid only during editing, each time a file containing generators is saved. There is no overhead when executing the program.

Controllability: Another benefit of using code generators for representing clones is that the propagation model between clone instances is clear and predictable. If needed, maintainers can understand more in-depth how clone instances are related by looking directly at the code generators in any text editor. For instance, the condition for adding statements in some instances of a type-3 clone are clearly visible in the corresponding code generator. At the extreme, maintainers can even take complete control over the clone refactoring by directly adjusting the code generators.

V. Refactoring clones with code generators

As explained in the previous section, the first step of our hybrid method consists in refactoring clones using code generators. This section explains how this can be done based on the information provided by any clone detector and using a suitable code generation technology. We first present the code generation technology before explaining how the selected clones have been refactored.

A. Code generation with Metapp

In C/C++, the native way to perform source code generation is using macros and conditional compilation, as provided by the cpp preprocessor. The main expressiveness limitation of these mechanisms is that they are not compositional, as a macro cannot contain #if directives. Instead, we use a flexible open-source preprocessor called Metapp[2] adding textual code generators to any programming language [13], in the spirit of the cpp preprocessor but with some important advantages, among which: macros and conditional code generation can be freely composed with each other; any Perl expression can be used in conditional and looping code generation statements; any Perl command can be executed at code generation time, for instance to set meta-variables, invoke Perl subroutines, etc.

When parameterized for C/C++, the Metapp preprocessor adds the following minimal set of code generation constructs to C/C++ programs, taking the form of stylized comments:

`//#copy name(...)`: invokes the code generator called $name$ passing it the given parameter values and outputs the source code produced by it. The code generator must be placed in a file called $name$.

`//#bind $param1, $param2, ...`: within a code generator, declares formal parameters named $param1$, $param2$, ..., that are to be bound to actual values passed at invocation of the code generator (via the "#copy" statement).

`//#if expr ... //#else ... //#fi`: Evaluate the Perl expression $expr$. If true, process the source lines between the "#if" and the "#else" constructs; otherwise, process the source lines between the "#else" and the "#fi" constructs.

`//#while expr ... //#end`: Repeatedly process the lines between the the "#while" and the "#end" constructs as long as the Perl expression $expr$ evaluates to true.

`//# perlcmd`: Evaluates the Perl command $perlcmd$ (there must be a space between the "//#" prefix and the command). This construct is mostly used for setting meta-variables with a Perl command of the form "$x=value".

[2]http://www.metaware.fr/metapp

```
1 //#bind $type
2 operator $type () {
3   decNumber decNum,intDecNum;
4   int32_t thisScale=precision;
5 //#if $type eq "char" || $type eq "bool"
6   $type i;
7 //#else
8   int32_t i;
9 //#fi
10  decContextZeroStatus(&set);
11  decPackedToNumber(this->data, sizeof(this->data),
12                                &thisScale, &decNum);
13  decNumberToIntegralExact(&decNum, &intDecNum, &set);
14  i=decNumberToInt32(&intDecNum, &set);
15  return i;
16 }
```

Fig. 1. Code generator "operator_int" unifying a type-3 clone.

Code generation constructs must appear at the beginning of a line (possibly after some whitespace). Any line not starting with the "//#" prefix is considered a source C/C++ line, and is output as is. However, source lines may contain Perl meta-variables of the form "$name", which are substituted with their current values.

B. Writing the code generators

It is very easy to refactor various types of clones using Metapp code generators, as follows.

Type-1 clones are simply represented by code generators with no parameters, which in fact contain only standard C/C++ code. This trivial sort of code generators could also be implemented with a standard C "#include" statement.

Type-2 and type-3 clones are represented by parameterized code generators not containing, and respectively containing, conditional code generation. For instance, the code generator in Figure 1, placed in a file called "operator_int" unifies several instances of a type-3 clone that represent methods for converting a decimal number to different integral types. The generator declares on line 1 a parameter called "$type" that is used a first time on line 2 as the name of the C++ conversion operator, and two more times on line 5 in a conditional construct to generate one of two possible declarations for variable "i". As can be seen, a code generator not only captures the variations between instances of a type-3 clone, but also relates the variations to the value of its parameters. This conditional declaration of a local variable is one of the clone variations that cannot be easily expressed with C++ templates, but are easy to express with code generation. The corresponding clone instances (the conversion operators to integral types) are replaced by calls to this code generator, as shown in Figure 2.

The correctness of this refactoring can be checked by preprocessing the refactored code with Metapp, and verifying that this produces the original program, using a standard comparison tool such as "diff" under Unix. Of course, variations in indentation and comments may be tolerated in order to allow unifying clones with unimportant differences.

Following this method, we manually expressed all the 17 selected clones as 17 code generators, and we replaced 110 instances of these 17 clones with calls to the generators. Then, we certified by iso-generation that the refactoring was correct.

```
...
/*-------------------------------------*/
/** cast operator to a char            */
/*-------------------------------------*/
//#copy operator_int("char")

/*-------------------------------------*/
/** cast operator to a bool            */
/*-------------------------------------*/
//#copy operator_int("bool")

/*-------------------------------------*/
/** cast operator to an int            */
/*-------------------------------------*/
//#copy operator_int("int")
...
```

Fig. 2. Excerpt of the refactored file "decimal.h".

VI. Using the editor

The refactored file "decimal.h" in Figure 2 can be edited using a standard C/C++ editor. Due to the encapsulation of code generation constructs in stylized comments, they will not trigger any parsing errors. However, all clone instances will figure as "//#copy" constructs, which means that maintainers will have to open the file containing the corresponding code generator to see the code to be generated. This is one part of the cognitive cost of using the refactored abstraction. Another part of the cognitive cost is that a file such as "operator_int" above contains code generation tags such as "//#bind" and "//#if" and meta-variable references, that maintainers have to master for understanding the code generated for each instance, and for eventually modifying this code.

The goal of the Stereo editor is to shield maintainers from these abstraction costs, by presenting them the original, unfactored C/C++ code, enriched with clone information, and by enabling them to change the code generators in a very intuitive way.

Thus, instead of opening in a standard C/C++ editor file "decimal.h" in Figure 2, maintainers open in the Stereo editor the generated code "pp/decimal.h" in Figure 3 produced by the Metapp preprocessor from the former file. The latter file is identical to the original code before clone refactoring, except that clone instances are clearly shown using a distinctive background color (light blue) and are initially not modifiable, which means that any editing action in these blocks, such as inserting or deleting a character, will fail. To inform maintainers about the provenance of each clone instance, two kinds of generation marks are introduced in the code:

- *line marks*: lines consisting in a "#line" tag, indicating the source file and line number from which the subsequent block of lines has been produced
- *meta-variable marks*: inline comments of the form "/*#<$$var$*/ val /*#>*/", where var is a meta-variable name which has been substituted with value val in the generated code.

These marks are shown in another distinctive color (gray) to clearly mark the fact they are added by the system. Marks can never be modified in the Stereo editor.

Helped by the coloring and marks in the code, maintainers not only know which parts of the code are in a clone and the name of the clone (that is, the name of the generator refactoring the clone), but also see variations between instances, as indicated by the meta-variable marks.

Thus, in the initial state of the editor, maintainers can edit any part of the code not belonging to a clone. We call this editing mode the editing of the "main" file. For modifying a clone instance, maintainers must double-click on that instance. This puts the editor in a new state, in which that instance becomes editable, and all the rest of the file becomes uneditable, as shown in Figure 4. This is reflected in the editor window by de-coloring the instance (*i.e.* using the default background color for it) and coloring the rest of the file using the uneditable background color. The coloring of marks is not changed, as these can never be edited.

When in this editing mode of an instance, maintainers can modify any part of the instance except the marks. Saving the file has the effect of updating *all* the instances of the current clone in the same way and switching back to the default editing mode of the main file. (The precise meaning of updating the other instances "in the same way" will be detailed in the next section.) Alternatively, the modifications done on the instance can be cancelled (by hitting the Escape key).

If the maintainers need to modify a clone instance independently of its siblings, they can *decouple* it from the generator, using a contextual menu. The decoupling action must be performed while in the editing mode of the main file. As a result, the instance is de-colored and all the inner marks are removed; the text in the instance thus becomes editable as an integrating part of the main file. This feature is very useful in a common clone-based programming practice in which one or several copies of a code fragment are first created by cloning and adaptation; then, during some time, the copies must inherit any modification of the original fragment; at some later point, the decision is taken to evolve one of the copies in an independent way to prototype some new features incompatible with the original code, or implying too many differences. Being able to decide this with a simple gesture is essential for the usability of a clone-aware editor.

VII. Implementation

The Stereo editor is implemented as an Eclipse plugin extending the standard plugin for editing C/C++ code, called CDT (C/C++ Development Tool). Thus, the Stereo editor features are added to all the existing IDE features, such as syntactic coloring or various forms of code analysis and error reporting.

The central idea of the implementation is that when modifications to a generated file are saved, the modifications are propagated back by the editor into the corresponding source file and code generators. Then, the updated source file is preprocessed again with the updated code generators, with the effect of updating all the clone instances in the editor. The back-propagation process is non-trivial for several reasons. Firstly, depending on the part of the generated file that has been modified, changes must impact either the main source file or the code generators called by it. The editor has to figure out which one(s) must be updated. Secondly, modifications cannot be propagated line by line, because they may consist not only in changing existing lines, but also suppressing lines or adding

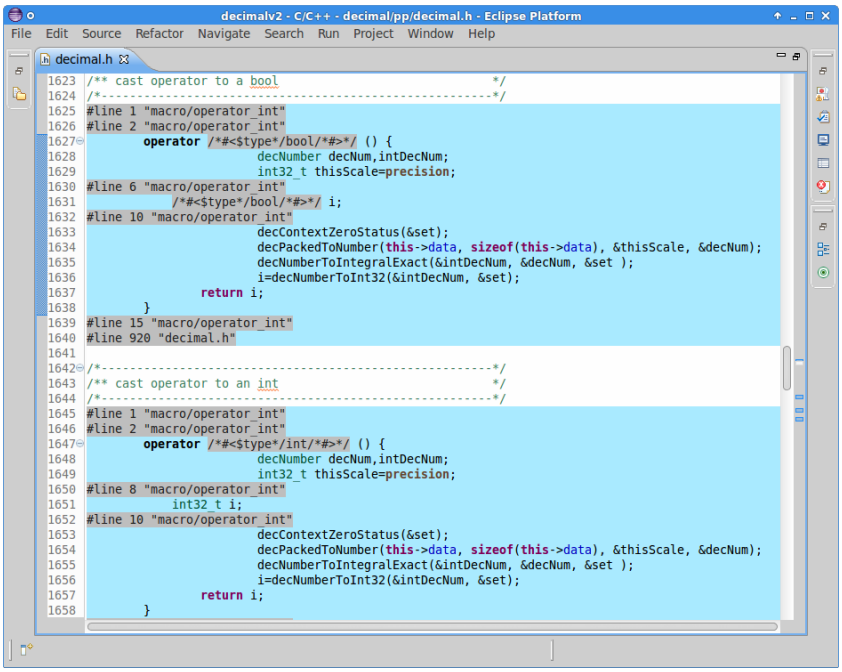

Fig. 3. Editing the main file “pp/decimal.h”.

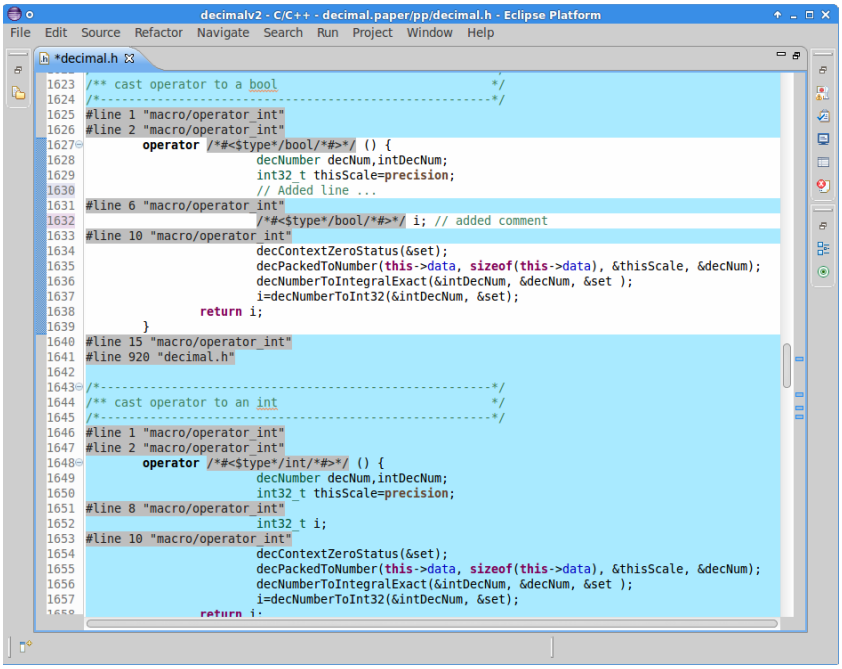

Fig. 4. Editing an instance within “pp/decimal.h”.

new lines. Anyways, line numbers are not the same in the generators and the generated files, because preprocessor lines are interpreted out in the generated file. Thirdly, modifications may also impact the structure of the code generators. Thus, when a clone instance is decoupled from its generator, the call to the generator must be removed in the source file and replaced with the generated, and possibly modified code.

For addressing these issues, the back-propagation of changes in our implementation is guided by the marks introduced by Metapp. More precisely, let us assume that one opens the generated file F' produced by the Metapp preprocessor from the refactored file F. Metapp interprets the code generation constructs in F, and the generated file F' will thus contain parts identical to those in F (modulo meta-variable substitutions) and generated lines coming from other files $(G_j)_{j=1..n}$ implementing the code generators called, directly or indirectly, from within F. The origin of each source line in F' and the meta-variable substitutions that were performed are indicated by the marks introduced by Metapp. These marks

are used by the editor to colorize generated code and to propagate modifications of F' back into the source files F and $(G_j)_{j=1..n}$, as follows.

A. Coloring

In the default editing mode of the main file, the lines originating from code generator files $(G_j)_{j=1..n}$ are colored as uneditable. By using the mechanism of Eclipse listeners, editing events such as keystrokes or mouse-based manipulations are intercepted by the Stereo editor. If an event would have the effect of modifying an uneditable section or a mark, it is discarded by the editor. Other events are delivered to the base CDT editor for their usual handling, and coloring is recomputed and re-applied if necessary. When entering the editing mode of an instance by doubly-clicking inside the instance, the same mechanisms are used to de-color the lines of that instance and color as uneditable the rest of the file. The code generator G_i corresponding to the instance is identified from the position of the double-click and the generation mark above it. Note however that not all the lines in F' originating from G_i are part of that precise instance, because a code generator is usually called multiple times in a file. Only the lines coming from G_i around the place where the double-click occurred are made editable. For instance, in Figure 4, only the top instance of "operator_int" is made editable. The start and end of an instance can be recognized using their line number equal to 1, respectively to the last line in the generator. Editing one instance at a time is important for ensuring that no conflicting modifications can be done on different instances of G_i. Note also that the lines constituting an instance of G_i can be non-contiguous in the view, if the generator G_i calls itself other generators (not shown in the figure).

B. Saving changes

The action of saving changes on F' must thus update a single file, which is, depending on the editing mode, either the code generator G_i corresponding to the edited instance, or the main file F. Without restricting the generality, let us assume the instance editing mode, in which file G_i has to be saved. This file must be updated to take into account all the editable lines in the editor window, as they potentially have been changed since the editing mode of the instance began. The difficulty is that the expanded file F' in the editor window does *not* contain all the lines in G_i, for two reasons. Firstly, all the code generations constructs in G_i have been interpreted out and only the result of executing them can be found in F'. Secondly, even some source lines in G_i may have disappeared in F' if they were in some non-selected branch of a conditional generation construct.

For instance, when comparing the top instance in Figure 4 with its generator "operator_int" in Figure 1, one can see that all the code generator constructs are missing in the instance; the declaration "int32_t i" (line 8 in Figure 1) is also missing because this branch was not selected in this instance.

Both kind of missing lines from the generator G_i should be retrieved directly from file G_i. We implemented a merging algorithm that combines the generator file G_i and the edited instance in the editor window, from F', based on the following rules: **(R1)** all generator construct lines are taken from file G_i; **(R2)** whenever a source code block exists in the edited instance in F', it is taken from F', because is may have been updated; if the source code contains meta-variable values, they are substituted back with the meta-variable name taken from the meta-variable mark; **(R3)** the other source code blocks are taken from the generator file G_i.

Note that the merging algorithm does not proceed one source line at a time, but rather one source block at a time, because source lines may have been added or deleted in F', so the source blocks in F' may contain more or less lines than the corresponding source blocks in G_i.

For instance, when saving changes to the edited instance in Figure 4, the instance in the editor is merged with the code generator "operator_int" in Figure 1 as follows: lines 1, 5, 7, and 9 in Figure 1, representing code generation constructs are taken from the code generator (rule R1); the 3 source blocks beginning in the instance at lines 1627, 1632, and 1634 are taken from the editor, with their updated contents (rule R2); the source block missing in the instance is taken from the code generator (line 8 in Figure 1) (rule R3). The merged contents is then written to the disk file "operator_int".

In order to report changes on all the other instances, the editor triggers a regeneration of the code by invoking the Metapp preprocessor on file F thereby recreating the file F' shown in the editor window, and switches back to the default editing mode. As a result, all instances of G_i appear updated.

C. Decoupling

The decoupling action is implemented by replacing a call to a code generator "//#copy G_i(...)" in file F with the generated code taken from F', in which any marks have been stripped away. The decoupling action can only be done in the default editing mode, in which file F' may have been modified with respect to file F, so a merging is necessary when the file is saved. The same merging algorithm described above is used to merge F with F', with only one slight change: as an exception to rule R1, the code generation construct "//#copy G_i(...)" line within F corresponding to the instance is not copied from F, but rather replaced with the instance source code from F', with marks stripped.

Some subtleties arise when the code generator G_i calls itself other code generators G_k. In that case, we adopted the simplest strategy that consists in decoupling all the embedded instances of G_k at the same time. Another, more complex, option would have been to allow maintainers decoupling such embedded instances one level at at time.

VIII. Evaluation

Maintainers are shielded from the abstraction costs related to the code generators *as long as* they can do their maintenance tasks using the Stereo editor, without looking to the refactored code. However, not all maintenance task can be done via the Stereo editor.

More precisely, the tasks that are possible in the Stereo editor involve modifying any number of source code blocks delimited by code generation constructs: source blocks in the main file can be modified in the default editing mode; source

blocks in a code generator can be modified while editing an instance that contains that block.

Modifications that *cannot* be done using the Stereo editor are those that involve modifying the code generator constructs themselves: deleting or modifying existing constructs, or adding new constructs. For instance, changing the parameters passed to a code generator cannot be done as it would involve modifying the values in a "//#copy" construct. As another example, adding a new parameter to a type-3 clone in order to cover more clone instances cannot be done as it would involve changing the "//#bind" construct in the generator and all the calls to this generator. Of course, these modifications can all be done by opening the refactored code in a standard editor and adjusting the code generation constructs, but this involves paying the corresponding abstraction costs.

Taking this design limitation into account, it is important to estimate how many maintenance scenarios are covered by the Stereo editor in practice. A thorough answer to this research question will involve extensively using the Stereo editor for the maintenance of various projects, and measuring the ratio of maintenance task that can not be done in the editor.

As a first evaluation in this sense, we used the Stereo editor for taking over from its initial developers the maintenance of the Decimal class over a period of three months, starting with the version v1.3 that we refactored using code generators as presented earlier, and spanning 4 new releases of the class, up to the current version v1.7. The maintenance was rather intensive in response to the customer's request to clean up or otherwise justify the many code duplications, and in response to a few bug reports. Thus, the maintenance consisted in a total of 45 changesets, as summarized in Table I. Each changeset concerned one method or a set of similar methods (up to 24). For space reasons, we do not present all the 45 changesets individually, but rather group them into 9 categories, listed in column 1; column 2 gives the number of changesets in each category. The first 4 categories are self-explaining. Reparameterizing C++ templates involved restricting the types of some C++ template arguments that were overly general, and thereby were introducing unwanted overloaded operators. Unifying clone instances consisted in eliminating unnecessary variations between instances of some refactored clones, such as reordering common variable declarations in the same order for all instances; this had the effect of transforming some type-3 clones to type-2 clones, which are easier to understand and to maintain. Reducing code generators consisted in rewriting some of the clones, that were refactored as code generators, by using instead standard C++ mechanisms, such as calls to an ordinary C++ method, when we considered that the related abstraction costs were sufficiently low. Handling remaining TODOs consisted in replacing a few "TODO" comments with the appropriate handling code. Other refactorings included some code rewritings aimed at eliminating compiler or IDE warnings, or other idioms perceived as risky.

Due to the large number of clones in the code, and to the request to clarify the code duplications, many changesets (32 out of 45) impacted cloned code. Column 3 in the table gives the total number of clone instances concerned by the changesets in each category, with a total of 145 instances impacted by the 45 changes, and thus an average of 3.22 instances impacted per change. This already shows the usefulness of using our editor on this code, as all the instances in each clone could be modified at once. Note that some clones were impacted by different changesets, which explains that the total number of impacted instances (145) is superior to the total number of clone instances in the code (110).

TABLE I. MAINTENANCE ACTIONS POSSIBLE IN THE STEREO EDITOR.

Category	#change-sets	#instan-ces	#OK	#KO
Bug fixes	6	7	3	3
Drop unused variables	9	25	9	0
Drop unused methods	2	2	2	0
Standardize trace messages	7	20	7	0
Reparameterize C++ templates	6	60	6	0
Unify clone instances	4	16	4	0
Reduce code generators	3	12	3	0
Handle remaining TODOs	2	0	2	0
Other refactorings	6	3	6	0
Total	45	145	42	3

The measurement of our evaluation criterion (how often can changes be performed using the Stereo editor?) can be found in the last 2 columns of the table, respectively giving the number of changesets that could be done and could not be done using our editor. Overall, 42 changesets out of the 45 could be performed completely with the Stereo editor, which represents 93% of the cases.

Only 3 changesets (i.e., 7% of the cases) could not be performed in the Stereo editor. Two of these 3 changesets involved changing the values passed to a code generator, which involves modifying a "//#copy" construct. It should be possible to implement in the editor such changes by allowing maintainers to edit the value in a meta-variable tag of an instance, and propagating the change to the corresponding "//#copy" construct. The third changeset that could not be done in Stereo consisted in evolving some of the instances of a type-3 clone without decoupling them, but rather by introducing a new variation under a new "//#if" construct. This transformation could less easily be envisioned using our plugin.

Summarizing the experiment, the results are very encouraging, as in 93% of the cases the changes can be done within our editor. Only in 7% of the cases it was unavoidable opening the refactored code with code generators and adjusting this code directly. Even among these latter cases, it seems possible to automate a common case, that of changing the values passed to a code generator by manipulating an instance.

Thus, the Stereo editor was very effective in this experiment in terms of shielding maintainers from the costs of abstraction, and the the editor concept has the potential to isolate from the abstraction even more, if some extensions would be implemented. More experiments on a larger scale are needed in order to confirm these preliminary results.

This quantitative evaluation mainly concerned the expressiveness and abstraction cost improvements of our approach: the proportion of clones that can be effectively handled by our plugin, without paying the cost of abstraction. As we did not use an existing linked editor, we can only comment in a qualitative way the improvements in scalability and controllability. The back-propagation algorithm is linear in the size of the updated file, and the forward propagation by regenerating the code is linear in the size of the generated

file. Therefore, modifying a clone with 24 instances takes the same refreshing time as for a clone with 2 instances, and in all cases, this time is hidden within the native file saving time. The improvement in controllability is also apparent: when adding some lines to a clone instance, they are always added inside a particular block delimited by generation marks. It is clear that all clone instances containing that block will be updated the same way. This makes the propagation of changes clearly predictable, which eliminates the need to check updated instances manually. If developers want to know whether this block exists only in the current instance, in a group of instances, or in all instances, they may look to the code generator to see whether the block is conditionally included, and if so, under which condition.

IX. Related work

The related work closest to our approach are the different existing linked editors, as their user interface is very similar to ours: cloned code is colored in a special way, and edits of one clone instance are propagated to the other instances; this propagation can be inhibited if desired to evolve some instance independently. The main difference between our approach and classical linked editors lies in the representation of clone information. Linked editors store clone information as more or less opaque metadata separated from the program source, while our editor stores it in the code itself in readable form as code generators and calls to them. From this difference stem the better properties of our approach in terms of expressiveness, scalability, and controllability. More detailed comparisons with individual linked editors are provided below.

In Codelink [9], the user interactively selects and links N code fragments, and the editor highlights their variations in a different color. CodeLink handles type-1, type-2, and type-3 clones, but making no difference between the last two types: different parameter values and added/removed statements are all simply differences, as opposed to our code generators. The automatic differencing algorithm tries to report minimal differences, but its result cannot be manually improved when suboptimal. As opposed to that, code generators currently require refactoring the program manually, but provide complete control over the representation of differences. In terms of scalability, the propagation of changes in Codelink has exponential complexity $O(s^n)$ for n instances of size s, while our propagations are linear in the size of the code.

In CloneTracker [8], the user selects a clone detected by the SimScan clone detector, and the editor converts it to a Clone Region Descriptor, a persistent metadata higher level than offsets in source files. Clones described this way are limited to whole statement blocks. Simultaneous editing is possible only between 2 instances, and the propagations must be individually inspected by the maintainers, as the reported success ratio in around 80%. On the contrary, our code generators propagate changes to any number of instances in a predictable way, updating the corresponding source blocks, clearly marked in each instance. On the other hand, our editor currently lacks a refactoring function creating a first version of code generator based on the results of a clone detector.

CloneBoard [5] infers clone information from copy/paste operations in the editor. *After* modifying a clone instance, several "conflict resolutions" are proposed to maintainers, such as propagating the changes, adding parameters to a type-2 clone, adjusting the border of instances, or decoupling the instance. CloneBoard distinguishes type-2 and type-3 clones, but suffers from the same limitations in scalability. It would be interesting to incorporate into our editor the copy/paste monitoring feature as a source of clone information.

CSeR [11] is an Eclipse plugin tracking copy/paste clones, using an improved differencing algorithm that is based on parsed trees comparison rather then textual comparison. Thereby, it may give higher-level clone information, such as inferring the move of some program element instead of a delete and insert; it also distinguishes between type-2 and type-3 clones. The differencing algorithm is limited to 2 clone instances, which prevents scalability. CSeR is part of a complete system called PCM (for Proactive Clone Management) [18] whose plan was to integrate linked editing features.

Ideally, our editor should be complemented with an algorithm that automatically computes a code generator of a clone set, and should further be integrated with a plugin for interactive refactoring. For instance, there is some basic refactoring support in the standard Eclipse plugins for Java or C++, offering actions for extracting some piece of code as a separate method. More advanced support is provided in a recent work extending the CDT Eclipse plugin for C/C++ [27], able to unify N clones using a larger variety of existing language mechanisms, such as introducing conditional statements, or using templates in the extracted method. A similar work [28] extends the Eclipse plugin for Java with advanced automatic refactorings. Our approach is complementary to this line of work, as it could apply in the cases when the refactoring is not possible using the available language mechanisms. In those cases only, the code generators could take over the task and perform the refactoring in a transparent way. On the other hand, algorithms for automatically creating a code generator, expressed in a language called VCL, starting from a clone set were recently described [29]. However, they do not allow the back propagation of changes: if the user modifies the code, the clone detector must be run again, and the code generator recomputed. An integration with our approach would allow to keep the code generators and the generated code in sync, thus closing the round trip between the code generators model and a (partially) generated program.

X. Conclusion

We presented a new approach to the ubiquitous problem of handling duplications in source code. In our approach, the program must be first refactored using code generators. Currently this refactoring is manual, but this manual intervention does not incur any risks, as the correctness of the resulting program can be easily checked by textually comparing the original and the newly generated code. Then, special editor support is used to edit the refactored program while giving the illusion to maintainers that they interact with the original program, augmented with clone information, and ensuring the consistent modification of clone instances, when desired. Thereby, our approach offers nearly zero-cost abstractions with no refactoring risks, similarly to linked editors. However, in comparison to linked editors, our approach has better expressiveness, scalability, and controllability properties. A

relative downside is that some advanced maintenance scenarios requiring the restructuring of code generators can only be done by adjusting the generators directly, in any other editor; this is easily feasible, but requires paying in these cases the cost of abstraction (understanding and changing the generators). We have shown on a first small-size but real-world experiment that this kind of scenarios is likely to occur quite rarely (7% of the cases in our experiment).

As a consequence, we believe that this editor concept constitutes a first step on a promising path for managing code duplication in a scalable way and without the risks usually associated to such a massive change.

Our approach can also be envisioned as a complement to existing clone refactoring techniques, to be used for the clone sets that cannot be easily and safely expressed by the mechanisms available in the language.

In the future, beyond implementing the already mentioned optimizations, we plan to apply the concept for editing other languages than C/C++ and using other code generation technologies than Metapp; this should be feasible by introducing the needed marks during code generation and propagating changes from an instance to its generator.

ACKNOWLEDGMENTS

Thanks to Papa Ousmane Diouf for implementing the Stereo plugin for Eclipse, by efficiently solving many technical issues. Special thanks to Nicolas Anquetil and Stéphane Ducasse for their most useful comments on an earlier draft.

REFERENCES

[1] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. 2004. *Aries: Refactoring support environment based on code clone analysis*. The 8th IASTED International Conference on Software Engineering and Applications (SEA 2004). 222-229.

[2] L. Aversano, L. Cerulo, and M. Di Penta. 2007. *How Clones are Maintained: An Empirical Study*. In Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR '07). IEEE Computer Society, Washington, DC, USA, 81-90.

[3] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. *CCFinder: a multilinguistic token-based code clone detection system for large scale source code*. IEEE Trans. Softw. Eng. 28, 7 (July 2002), 654-670.

[4] R. Tairas and J. Gray. 2012. *Increasing clone maintenance support by unifying clone detection and refactoring activities*. Inf. Softw. Technol. 54, 12 (December 2012), 1297-1307.

[5] M. de Wit, A. Zaidman, and A. van Deursen. 2009. *Managing code clones using dynamic change tracking and resolution*. IEEE International Conference on Software Maintenance, 2009 (ICSM'09). 169 - 178.

[6] Semantics Design, Inc. *CloneDR (TM)*. http://www.semanticdesigns.com/Products/Clone/

[7] C. J. Kapser and M. W. Godfrey. 2008. *Cloning considered harmful" considered harmful: patterns of cloning in software*. Empirical Softw. Engg. 13, 6 (December 2008), 645-692.

[8] E. Duala-Ekoko and M. P. Robillard. 2008. *CloneTracker: Tool Support for Code Clone Management*. In Proceedings of the 30th ACM/IEEE International Conference on Software Engineering, May 2008, Formal Research Demonstration.

[9] M. Toomim, A. Begel, and S. L. Graham. 2004. *Managing Duplicated Code with Linked Editing*. In Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC '04). IEEE Computer Society, Washington, DC, USA, 173-180.

[10] J. R. Cordy. 2003. *Comprehending Reality " Practical Barriers to Industrial Adoption of Software Maintenance Automation*. In Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC '03). IEEE Computer Society, Washington, DC, USA, 196-.

[11] F. Jacob, D. Hou, and P. Jablonski. 2010. *Actively comparing clones inside the code editor*. In Proceedings of the 4th International Workshop on Software Clones (IWSC '10). ACM, New York, NY, USA, 9-16.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. 2007. *DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones*. In Proceedings of the 29th international conference on Software Engineering (ICSE '07). IEEE Computer Society, Washington, DC, USA, 96-105.

[13] N. Volanschi. 2012. *Safe clone-based refactoring through stereotype identification and iso-generation*. In Proceedings of the 6th International Workshop on Software Clones (IWSC '12). IEEE Press, Piscataway, NJ, USA, 50-56.

[14] S. Jarzabek and S. Li. 2006. *Unifying clones with a generative programming technique: a case study*. Practice Articles. J. Softw. Maint. Evol. 18, 4 (July 2006), 267-292.

[15] H. A. Basit, U. Ali, and S. Jarzabek. 2011. *Viewing simple clones from structural clones' perspective*. In Proceedings of the 5th International Workshop on Software Clones (IWSC '11). ACM, New York, NY, USA, 1-6.

[16] R. C. Miller and B. A. Myers. 2002. *LAPIS: smart editing with text structure*. In CHI '02 Extended Abstracts on Human Factors in Computing Systems (CHI EA '02). ACM, New York, NY, USA, 496-497.

[17] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. 2014. *Towards User-Friendly Projectional Editors*. In Proceedings of the 7th International Conference on Software Language Engineering (SLE'14), Vsters, Sweden, September 15-16, 2014. LNCS vol. 8706. Springer. 41-61.

[18] D. Hou, F. Jacob, and P. Jablonski. 2009. *Exploring the design space of proactive tool support for copy-and-paste programming*. In Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '09), Patrick Martin, Anatol W. Kark, and Darlene Stewart (Eds.). IBM Corp., Riverton, NJ, USA, 188-202.

[19] M. F. Zibran and C. K. Roy. 2012. *The Road to Software Clone Management: A Survey*. Technical Report 2012-03, Department of Computer Science, The University of Saskatchewan, Canada, February 2012, 62 pp.

[20] Simon Harris. *Simian - Similarity Analyser*. http://www.harukizaemon.com/simian/

[21] G. P. Krishnan and N. Tsantalis. 2014. *Unification and Refactoring of Clones*. IEEE CSMR-WCRE 2014 Software Evolution Week (CSMR-WCRE'2014), Antwerp, Belgium, February 3-7, 2014. 104-113.

[22] S. Schulze, M. Kuhlemann, and M. Rosenmller. 2008. *Towards a refactoring guideline using code clone classification*. In Proceedings of the 2nd Workshop on Refactoring Tools (WRT '08). ACM, New York, NY, USA. Article 6, 4 pages.

[23] L. Yu and S. Ramaswamy. 2008. *Improving modularity by refactoring code clones: a feasibility study on Linux*. SIGSOFT Softw. Eng. Notes 33, 2, Article 9 (March 2008), 5 pages.

[24] Z. Chen, M. Mohanavilasam, Y. W. Kwon, and M. Song. 2017. *Tool Support for Managing Clone Refactorings to Facilitate Code Review in Evolving Software*. In 41st IEEE Annual Computer Software and Applications Conference (COMPSAC'17), Turin, pp. 288-297.

[25] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. 2005. *Beyond templates: a study of clones in the STL and some general implications*. In Proceedings of the 27th international conference on Software engineering (ICSE '05). ACM, New York, NY, USA, 451-459.

[26] V. Saini, H. Sajnani, J. Kim, and C. Lopes. 2016. *SourcererCC and SourcererCC-I: tools to detect clones in batch mode and during software development*. In Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16). ACM, New York, NY, USA, 597-600.

[27] K. Narasimhan and C. Reichenbach. 2015. *Copy and Paste Redeemed (T)*. In 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, pp. 630-640.

[28] D. Mazinanian, N. Tsantalis, R. Stein and Z. Valenta. 2016. *JDeodorant: Clone Refactoring*. In 38th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C), Austin, TX, pp. 613-616.

[29] H. A. Basit, H. S. Khan, F. Hamid and I. Suhail. 2015. *Tool support for managing method clones*. In 9th IEEE International Workshop on Software Clones (IWSC), Montreal, QC, pp. 40-46.