Does Cloned Code Increase Maintenance Effort?

Manishankar Mondal Chanchal K. Roy Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{mshankar.mondal, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—In-spite of a number of in-depth investigations regarding the impact of clones in the maintenance phase there is no concrete answer to the long lived research question, "Does the presence of code clones increase maintenance effort?". Existing studies have measured different change related metrics for cloned and non-cloned regions, however, no study calculates the maintenance effort spent for these code regions.**

In this paper, we perform an in-depth empirical study in order to compare the maintenance efforts required for cloned and non-cloned code. For the purpose of our study we implement a prototype tool which is capable of estimating the effort spent by a developer for changing a particular method. It can also predict effort that might need to be spent for making some changes to a particular method. Our estimation and prediction involve automatic extraction and analysis of the entire evolution history of a candidate software system. We applied our tool on hundreds of revisions of six open source subject systems written in three different programming languages for calculating the efforts spent for cloned and non-cloned code. According to our experimental results: (i) cloned code requires more effort in the maintenance phase than non-cloned code, and (ii) Type 2 and Type 3 clones require more effort compared to the efforts required by Type 1 clones. According to our findings, we should prioritize Type 2 and Type 3 clones when making clone management decisions.

I. Introduction

During the evolution of a software system, the frequent copy paste activities performed by the programmers cause multiple copies of exactly or nearly similar code fragments to co-exist in the code base. According to the literature, these same or nearly similar code fragments are termed as clones. Beside the copy-paste activities some other factors such as unrealistic deadlines of projects, programmer behaviours like laziness and tendency to repeat common solutions, technology limitations, code evolvability, code understandability and external business forces have influences on code cloning [13]. Whatever may be the reasons behind cloning, the impacts of clones are of great importance from the perspective of software maintenance.

On the basis of a number of existing investigations clones have mixed impacts in the maintenance phase. The studies in favour of clones argued that clones are not harmful [2], [10], [14], [15], [29] instead clones can be helpful from different perspectives [13]. On the other hand, there are empirical evidences that clones can introduce temporarily hidden faults [11], bugs [20] and unintentional inconsistent changes [3], [4], [9] to the software system. A number of studies [11], [22]–[25] also show that clones can exhibit higher instability than non-clone code in the maintenance phase.

Motivation. Focusing on the negative impacts of code clones researchers suspect that code clones can possibly increase software maintenance effort and costs. However, there is no empirical evidence regarding this. By analyzing the negative impacts of code clones we see that each of these leads to an increased modification to the source code. Fixation of a previously introduced bug or fault or propagation of a particular change to ensure consistency ultimately result in additional source code changes as well as efforts in the maintenance phase. However, bugs or faults can also be introduced in the non-cloned code. So, by measuring the source code change efforts of cloned and non-cloned code in the maintenance phase and comparing these efforts we can determine whether clones can really increase maintenance efforts compared to non-cloned code or not. Unfortunately, none of the existing studies have compared efforts required for cloned and non-cloned code. We identify the following issues of the existing clone impact studies.

(1) Existing studies have only quantified the amount of modifications in the source code without considering how much time or effort has been spent for understanding prior to a particular change. There are many situations where most of the time regarding a particular change is spent for understanding where to make that change and how to make that change. A little amount of change might require a considerable amount of understanding time if the responsible programmer is new to the particular project. Thus, only the amount of lines or tokens changed in a code region (cloned or non-cloned) can never be a good estimator of the total effort spent for that region.

(2) Most of the studies did not compare the impacts of different types of clones. Comparison among clone-types is important because, if a particular clone-type requires more effort than the other types, we can suggest programmers to avoid that particular clone type or to take extra care of it.

A number of effort estimation models such as: COCOMO [5], function point based model [1], use case based models [18], and analogy based model [19] currently exist. However, such models cannot be used to calculate the efforts required for cloned and non-cloned code of a software system separately.

Contribution. Focusing on the above issues we perform an empirical study for determining and comparing the maintenance efforts of cloned and non-cloned code. Our study involves: (i) estimation of efforts for already happened changes to a particular method, and (ii) prediction of efforts that might need to be spent for changing a particular method. We implement our estimation and prediction procedures being inspired by the effort estimation models [1], [5], [18], [19]. While our estimation might not be 100% accurate, it gives an overall idea of the effort. Also, our primary goal is to compare the maintenance effort required for cloned and non-cloned code. We believe that our implementation can be used for this purpose. We emphasize two things in our implementation: (i) amount of source code (in terms of tokens) that has been (or might be) changed in a particular method, and (ii) amount of source code that might need to be understood for making changes to a particular method. For determining which other methods we need to understand for changing a particular method we extracted method co-change information by mining the evolution history of the candidate software system.

Findings. We applied our implementation on hundreds of revisions of six open source software systems of diverse nature covering three programming languages (Java, C and C#) and estimated the maintenance efforts of cloned and non-cloned code. According to our analysis: *Cloned code requires more effort in the maintenance phase than non-cloned code. Type 3 and Type 2 clones require more effort compared to Type 1 clones.* Our findings imply that cloned code can often increase software maintenance costs compared to non-cloned code. When taking clone management decisions we should primarily focus on Type 3 and Type 2 clones. Our effort estimation tool can always be helpful to the managers as well as programmers in determining and predicting source code change efforts.

The rest of the paper is organized as follows: Section II describes our effort estimation technique, Section III discusses co-changed method groups, Section IV describes the experimental steps, Section V contains experimental results and analysis, Section VI evaluates the usefulness of our procedure for predicting code change effort, Section VII discusses possible threats to validity, Section VIII discusses the related word, and Section XI concludes the paper by mentioning future work.

II. Effort Estimation Procedure

We implement our effort estimation procedure emphasizing two things: (i) calculation of effort required by the changes which have already taken place, and (ii) prediction of effort that might be required for a future change in a particular method. There are several factors such as type and complexity of the candidate project, programmer expertise, effort for finding a bug or fault, software testing effort and so on which have not been considered in our study. There are two reasons why we have not considered these - (i) determination of project type and programmer expertise requires manual investigation and also, as we are comparing the efforts for cloned and non-cloned code these factors will be neutralized, and (ii) calculation of bug finding and testing efforts cannot be fully automated. Our effort estimation procedure is described below.

A. Calculating Effort for Changes that Occurred in a Method

For calculating effort spent for changes that previously occurred in a particular method we need to calculate two types of efforts - (i) understanding effort, and (ii) change effort.

Understanding effort. This type of effort is spent for understanding relevant methods. We need to spend time for understanding the following five types of methods:

(1) The particular method to be changed.

(2) The user defined methods called from this method.

(3) The user defined methods which have called this particular method.

(4) The methods which contain some fragments that are cloned with some fragments of this particular method. We need to understand the cloned methods because these methods might need to be changed to ensure consistency of cloned fragments.

(5) The methods that have high probability of being co-changed with this particular method. Co-changed methods (explained in Section III) are those methods that are changed in a single commit operation to achieve a goal. Generally, co-changed methods implement different functionalities of a particular goal.

Change effort. This type of effort is spent for making changes to the particular method and some other methods which are cloned or co-changed with that particular method. Calculation of change effort involves measuring the followings.

(1) Number of tokens changed in the particular method.

(2) Number of tokens changed in the co-changed methods.

(3) Number of tokens changed to ensure consistency in the cloned portions of other methods that are cloned with this particular method.

B. Predicting Effort for Future Changes in a Method

Prediction of effort for changing a particular method in future includes the understanding efforts (described in Section II-A) for this method. We also need to predict the number of tokens that might need to be changed in the particular method as well as in the methods that are co-changed (i.e., changed together) with it. We do not predict the number of tokens to be changed in the cloned methods because we do not know whether the changes will take place to the cloned or non-cloned portions of this particular method.

Predicting the number of tokens to be changed is tricky. Suppose, we want to predict the efforts for changing a particular method m which has been co-changed with several other methods during evolution. To predict the efforts for changing m we also need to predict the efforts for changing the co-changed methods. Suppose, m_{co} is a method which has been co-changed with m. We need to predict the followings.

- How many tokens might need to be changed in the particular method m.
- How many tokens might need to be changed in the co-changed method m_{co}.

We predict the number of tokens that might need to be changed in m by determining the average number of tokens changed in m per revision. The revisions that we consider for this case are those where m has received some changes. If m has been changed in r number of previous revisions during evolution, and the total number of tokens (of m) changed (added, deleted or modified) in these r revisions is T, we predict the number of tokens in m that might need to be changed in the current revision by Eq. 1.

$$T_{predict} = T \ / \ r \tag{1}$$

Here, $T_{predict}$ is the predicted number of tokens to be changed in m. For predicting the number of tokens to be changed in the co-changed method m_{co} we calculate:

- The average number of tokens that have been changed in m_{co} per revision (we consider only those revisions where m_{co} has changed) during the evolution
- The probability by which m_{co} co-changes with m.

Suppose, the total number of revisions where m_{co} has been changed is r_{co}, the total number of tokens changed in m_{co} in these revisions is T_{co}, the total number of revisions where m has been changed is r and the total number of revisions where both m and m_{co} have been changed is R_{co}. We predict the number of tokens to be changed in m_{co} using Eq. 2.

$$T_{predict\ co-change} = \frac{T_{co}}{r_{co}} \times \frac{R_{co}}{r} \tag{2}$$

Here, $T_{predict\ co-change}$ is the predicted number of tokens to be changed in m_{co}. Two quantities are multiplied in the above equation. The first quantity is the average number of tokens changed so far in m_{co} and the second quantity is the probability of m_{co} to be co-changed with m.

C. Effort Calculation and Prediction

Suppose we are calculating the maintenance effort required for some changes that occurred in a method in a particular revision of a software system. Before the occurrence of changes, the method had T tokens. The user defined methods called from this method had T_{called} tokens and the user defined methods calling this method had $T_{calling}$ tokens. Also, the unique co-changed method groups (explained in Section III) excluding this method had T_{co} tokens in total. The methods that were cloned with this method but not in co-changed method group had T_{cloned} tokens. Because of the changes, T_c tokens were added in, modified or deleted from cloned portions of this method. In the same way, T_n tokens were added in, modified or deleted from non-cloned portions of this method. T_{coc} tokens were changed in all other methods in the co-changed groups. $T_{clonedc}$ tokens were changed in the cloned portions of the cloned methods (that are not in the co-changed group) to maintain consistency. The maintenance effort for the changes that occurred in this particular method can be calculated according to the following equations.

$$UE = (T + T_{called} + T_{calling} + T_{co} + T_{cloned}) \times a \times E \quad (3)$$

$$TCE = (T_c + T_n + T_{coc} + T_{clonedc}) \times b \quad (4)$$

$$TE = UE + TCE \quad (5)$$

Here, UE, TCE and TE are respectively the understanding effort, token change effort and total effort. All the operands in the above equations excluding a, b and E are quantifiable by examining the revisions of the subject system. The constant a is the effort to understand a single token. Constant b is the effort to add, delete or modify a single token. E quantifies the expertise of the responsible programmer. The value of E spans between 0 and 1. E = 1 means that the programmer is totally new in changing the particular method and E = 0 means that the programmer is expert enough in changing this method, and thus, he / she does not need understanding effort. We can also separate the efforts spent for cloned and non-cloned code in the following way.

$$CE = UE + (T_c + T_{coc} + T_{clonedc}) \times b \quad (6)$$

$$NE = (T + T_{called} + T_{calling} + T_{co}) \times a \times E + (T_n + T_{coc}) \times b \quad (7)$$

Here CE is the effort spent for cloned code and NE is the effort for non-cloned code.

While predicting the efforts for changing a particular method we first predict the number of tokens to be changed in that method and in the co-changed methods using the equations Eq. 1 and Eq. 2 and then we predict the efforts according to the following equation.

$$PE = UE + (T_p + T_{pco}) \times b \quad (8)$$

Here PE is the predicted effort, T_p is the probable number of tokens that might need to be changed in the particular method and T_{pco} is the number of tokens that might be changed in the co-changed methods. We see that the predicted effort also includes the understanding effort UE.

D. Assignment of Values to The Constants

We can assign the values for a and b in terms of time but these values depend on several factors including type and complexity of the candidate project and expertise of the responsible programmer. The effects of these factors on a and b cannot be determined automatically. As our main focus is on the comparison of the efforts required for cloned and non-cloned portions, we do not need to know the exact values of a and b. Instead we need to assume the ratio between these two constants so that we can convert both the number of tokens to understand and the number of tokens to change to a common unit. We can then add the understanding and changing efforts of a particular code region (cloned or non-cloned) of a subject system to get the total effort for that region of the system. In our experiment, we select the ratio between a and b to be 3 (a/b = 3) as was assumed in a previous study [6]. We also consider E = 1 which means that the programmer who is responsible for changing a particular method has no prior knowledge about the method and other relevant methods. While it is true that the understanding efforts for $T_{called}, T_{calling}, T_{co}$, and T_{cloned} in Eq. 3 should be treated differently from the perspective of program comprehension, in our experiment we consider that the responsible programmer for making changes has no prior knowledge about the code-base. Thus, similar treatment of the understanding effort for $T_{called}, T_{calling}, T_{co}$, and T_{cloned} is reasonable.

III. Co-changed Method Groups

During the evolution of a software system multiple revisions of it are created. We denote the revisions by $revision(i)$ where $1 \leq i \leq n$. Here n is the total number of revisions of the software system created so far. A commit operation $commit(i)$ on $revision(i)$ causes the next revision $revision(i{+}1)$ to be created. For most of the cases a particular commit operation consists of several changes to the source code. An important fact here is that if the changes in a particular commit operation are not atypical [23], they are made to achieve a particular goal (or functionality) and hence, these changes are related. So, the methods in which these changes have occured are also related. These methods, that are changed in a particular commit operation, form a co-changed method group. While making some changes to a particular method, a developer should also be concerned about other methods in those co-changed method groups to which that particular method belongs.

Detecting Co-changed Method Groups. To determine the co-changed method group for a particular commit operation $commit(i)$ we accomplish several tasks - (1) detection of all methods in $revision(i)$ with corresponding beginning and ending line numbers, (2) determination of changes happened to $revision(i)$ with corresponding line numbers, (3) mapping these changes to the detected methods of $revision(i)$, and at last (4) retrieval of the changed methods. For a sequence of n revisions of a subject system we get a sequence of $n - 1$ commit operations. After discarding the commit operations with atypical changes we get our target commit operations. We discarded commits with atypical changes following the technique of Lozano and Wermelinger [23]. If the count of the target commit operations is t, we get t co-changed method groups. But, it is very likely that a particular co-changed method group will appear multiple times in this sequence

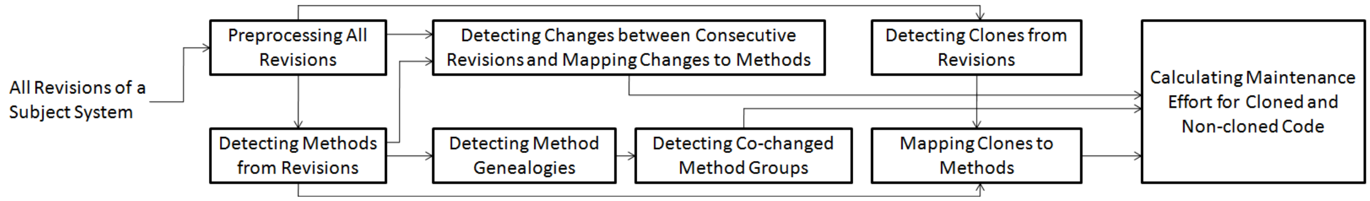

Fig. 1. Experimental steps and their interdependencies in calculating maintenance efforts for cloned and non-cloned code

TABLE I. SUBJECT SYSTEMS

	Systems	Domains	LOC	Revisions
Java	Ant-Contrib	Web Server	12,621	176
	Carol	Game	25,092	1699
C	Ctags	Code Def. Generator	33,270	774
	QMail Admin	Mail Management	4,054	317
C#	GreenShot	Multimedia	37,628	999
	Capital Resource	Database Management	75,434	122

TABLE II. NICAD SETTINGS

Clone Types	Identifier Renaming	Dissimilarity Threshold
Type 1	none	0%
Type 2	blindrename	0%
Type 3	blindrename	20%

of t groups because, changes in multiple commit operations might be centered around the same or similar goals. So, we need to determine the unique co-changed method groups for a sequence of commit operations.

Algorithm for detecting unique co-changed method groups: Suppose, we have already detected some unique co-changed method groups by examining some commit operations. We call this list of existing groups $existing\ list$. After getting a new group from the next commit operation, we first check whether this group is a proper subset of any group in the $existing\ list$. If this is true, we ignore this new group, otherwise we check the $existing\ list$ to find any group which is a proper subset of this new group. We discard these groups from the $existing\ list$ and add the new group to it. Then, we proceed with the next commit operation. However, at the very begining of this process (while examining the first commit), the $existing\ list$ remains empty.

IV. EXPERIMENTAL STEPS

We conduct our experiment on six subject systems listed in Table I. We download these systems from an on-line SVN repository[1]. The subject systems are diverse, differing in size, spanning six different application domains, and covering three different programming languages. We have implemented our effort estimation procedure as a tool using Java programming language with MySQL as the back-end database server and then applied the tool on each of the subject systems in Table I for calculating maintenance efforts of cloned and non-cloned code. The calculation has been done in the following steps as demonstrated in Fig. 1: **(1)** Preprocessing source code of each revision of a subject system by applying two preprocessing steps - (i) rearranging lines so that an isolated left or right brace (if a left or right brace remains in a line associated with no other character) gets deleted and added to the previous line (creating a blank line), and (ii) deleting blank lines and comments, **(2)** Detecting methods from each of the revisions of the candidate system by applying CTAGS[2], **(3)** Detecting method genealogies considering all the revisions of a subject system by applying the technique proposed by Lozano and Wermelinger [23], **(4)** Detecting code clones from each of the revisions by applying the NiCad clone detector [8], [27], **(5)** Mapping clones in each revision to the methods in that revision using their start and end line numbers, **(6)** Detecting changes between every two consecutive revisions using UNIX *diff*, and mapping these changes to the methods in these revisions, **(7)** Detecting co-changed method groups by analyzing the evolution history of the subject system, and **(8)** Calculating maintenance effort for cloned and non-cloned code.

For the details of the steps 2 to 6 we refer the interested readers to our earlier work [26]. Step 7 has been elaborated in Section III. Method genealogy detection step (i.e., Step 3) was essential for detecting the co-changed method groups. Step 8 will be elaborated below. Fig. 1 shows the interdependency among the eight experimental steps. An arrow from one rectangle to another means that the activity in the second rectangle depends on the output from the activity in the first one.

NiCad Setup. Using NiCad we detected block clones with a minimum size of 5 LOC. We used the NiCad settings in Table II for detecting three types of clones (Type 1, Type 2, and Type 3). For all the settings NiCad was shown to have high precision and recall [28].

Calculation of maintenance efforts. Suppose we are calculating the maintenance efforts for some changes occurred in a particular method in revision R. We need to determine two things: **(1)** the number of tokens that need to be understood for changing this particular method, and **(2)** the number of tokens that have actually been changed in this method and its co-changed methods. For determining these, we find whether there is any stored co-changed method group (in the database) to which this method belongs. We also find the methods which are cloned with this method by querying the clone detection result. In this way, we calculate the number to tokens to understand and the number of tokens actually changed for each of the methods changed in this revision. We calculate the efforts required for changing a particular method as well as for changing cloned code and non-cloned code according to the equations Eq. 5, Eq. 6 and Eq. 7.

In our experiment, we have not identified the caller-callee relationships among methods. These relationships are only necessary for understanding purpose. We detect co-changed method groups. By querying these groups we identify which methods need to be understood for changing which method.

[1]On-line SVN Repository: https://sourceforge.net/

[2]CTAGS: http://ctags.sourceforge.net/

TABLE III. EFFORTS REQUIRED FOR CLONED AND NON-CLONED CODE

Systems	Type 1			Type 2			Type 3		
	Ef_c	Ef_n	R	Ef_c	Ef_n	R	Ef_c	Ef_n	R
Ant-Cont.	0	1450.6	⊕	2259	1414.68	⊖	318	1496.95	⊕
Carol	1784.37	1327.40	⊖	2008.61	1351.97	⊖	1229.61	1165.19	⊖
Ctags	1720.62	1665.92	⊖	2006.78	1658.52	⊖	2365.09	1676.83	⊖
QMail Ad.	6780.39	4205.67	⊖	36993.6	3260.8	⊖	10852.85	2799.18	⊖
GreenShot	2806.31	3521.54	⊕	2879.92	3295.67	⊕	3121.93	3331.77	⊕
Capital Res.	0	344.27	⊕	0	344.27	⊕	1043.16	349.03	⊖

Ef_c= Effort Required for Cloned Code Ef_n= Effort Required for Non-Cloned Code

⊕= $Ef_c < Ef_n$ ⊖= $Ef_c > Ef_n$ R = Remark

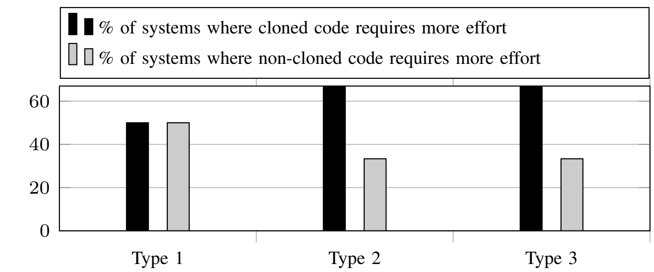

Fig. 2. Comparing efforts of non-cloned code and three types of cloned code

V. EXPERIMENTAL RESULTS AND ANALYSIS

We have applied our effort estimation tool on six open source subject systems and got the efforts for modifying cloned and non-cloned code. When considering code clones of a particular clone-type, the rest of the code-base (i.e., the code that we get by excluding code clones of that particular clone-type) was considered as the non-cloned code. The efforts are shown in the Table III. However, the efforts listed in the Table III are the summations of corresponding understanding and changing efforts. We call these efforts the *total efforts*. In the following subsections we provide our analysis on the total effort, understanding effort, and changing effort.

Overall analysis of experimental result. We can interpret Table III as consisting of 18 cases where each case consists of a particular subject system and a particular clone type. For each case, there are two efforts in the table - (i) effort for cloned code (Ef_c), and (ii) effort for non-cloned code (Ef_n) and a remark (⊖ or ⊕). The symbol '⊖' indicates that cloned code required more effort than the non-cloned code and '⊕' indicates the opposite. According to our result, for 61.11% cases (11 cases in total) cloned code required more effort than non-cloned code and the opposite is true for the remaining 38.89% cases. The difference between these percentages indicates that *cloned code requires more effort than the non-cloned code in general.* Obviously, our investigation is not exhaustive and the result might be skewed by selecting another set of subject systems. However, our investigation and experimental results provide an evidence that *cloned code often requires more effort in the maintenance phase than non-cloned code.*

We have also analyzed the results for each of the three types of clones individually and the findings have been presented using a bar graph (Fig. 2). From this graph we see that both Type 2 and Type 3 clones required more effort than non-cloned code for greater proportion of subject systems. According to this graph, *each of the three types of clones can require more effort than the non-cloned code during software evolution. The changes in Type 2 and Type 3 clones have higher probabilities of requiring more effort than those in Type 1 clones.*

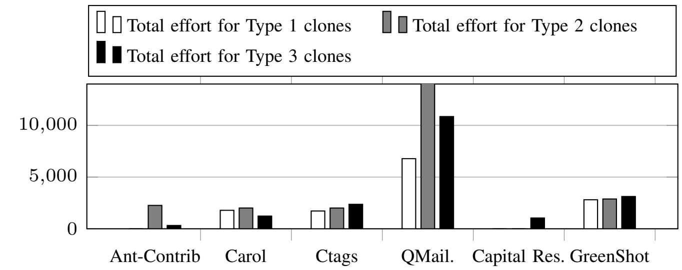

Fig. 3. Total effort for three types of code clones.

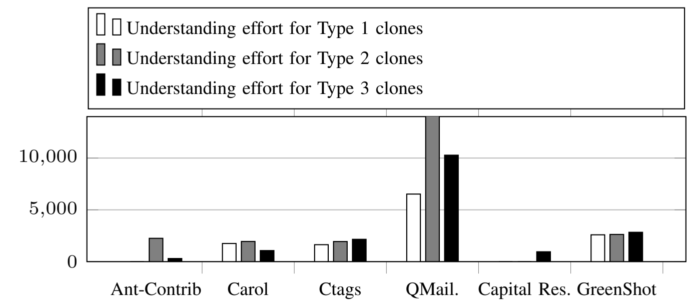

Fig. 4. Understanding effort for three types of code clones.

Overall Analysis Result: According to Table III and Fig. 2 we realize that cloned code can increase maintenance effort during software evolution compared to non-cloned code.

Type centric analysis of experimental result. As cloned code can increase maintenance efforts, this is important to analyze the variability of efforts required by different types of clones to identify which types of clones should be given more care. For this purpose, we have drawn the graph shown in Fig. 3 containing each of the six candidate systems with respective efforts for three clone-types. From this graph we see that for each of the three subject systems, Ctags, Capital Resource and Greenshot, the effort required for Type 3 clones was greater than the effort required for each of the other two types of clones. For the remaining three subject systems, the efforts required for Type 2 clones were the greatest ones. From this scenario we can come to the decision that *Type 2 and Type 3 clones generally require more effort than Type 1 clones.*

We also analyzed the understanding and modification efforts for each type of clones individually. For this purpose, we have drawn two more graphs in Fig. 4 and Fig. 5 showing the comparison of understanding and modification efforts for three types of clones respectively. Fig. 4 shows the comparison of understanding efforts for three clone-types. We see that understanding efforts exactly follow the total efforts (understanding + modification) (Fig.3). From this we decide that the total amount of efforts required for source code changes is mainly driven by the understanding efforts. From the other graph in Fig. 5 we see that for five subject systems excluding QMailAdmin, token change efforts for Type 3 clones were greater than those of the other two types of clones. We also see that for both cloned and non-cloned code, understanding efforts are always greater than modification efforts.

Type Centric Analysis Result: From the type centric analysis we learn that both Type 2 and Type 3 clones require more effort (both understanding and modification) than the effort

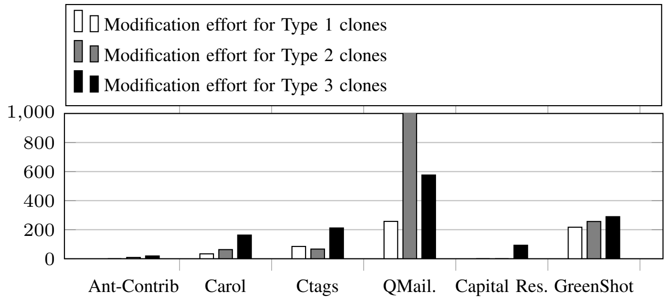

Fig. 5. Modification effort for three types of code clones.

required by Type 1 clones. Also, Type 3 clones require more modification efforts than the other two types. So, Type 2 and Type 3 clones should be given more care during development. More importantly, when taking clone management (such as clone refactoring or tracking) decisions we should primarily focus on Type 2 and Type 3 clones.

VI. Accuracy in Predicting Code Change Effort

We calculate the accuracy of our effort estimation tool in predicting code change effort in the following way. Suppose, the subject system on which we are working has R revisions in total. We consider a revision r which is not the most recent one ($1<r<R$). Suppose, several methods in this revision have been modified to create the immediate next revision. We calculate the actual number of tokens that were changed while changing these methods in revision r to create the next revision. We also use our tool to predict the number of tokens that might need to be changed in these methods in revision r using the co-change history of these methods as described in Section II-B. In this way, for all the revisions we have worked on, we calculate - **(i)** the count of actually changed tokens, **(ii)** count of tokens predicted to be changed, and **(iii)** the total count of methods changed. From these, we calculate the number of tokens actually changed per method and the number of tokens predicted to be changed per method. We calculate these two quantities for each of our candidate subject systems. We find the Pearson correlation between these two quantities.

The calculated values and the correlation result are shown in Table IV. From the table we see that there is a high correlation between these two quantities (the Pearson correlation co-efficient = 0.888754). If we disregard Capital Resource (it does not comply with the other five systems), the correlation coefficient becomes 0.9864. We also see that our tool is underestimating the count of tokens to be changed for most of the cases (except for Capital Resource). The reason behind this is explained in the following way.

For a particular method, while we predict the number of tokens to be changed in any one of its co-changed methods we multiply two quantities: **(i)** the number of tokens that is likely to be changed in the co-changed method, and **(ii)** the probability by which this method co-changes with the particular method. The probability becomes less than one if this co-changed method has not changed each time the particular method has changed in the past. This probability is underestimating the count of tokens to be changed in a co-changed method. We have seen that the count of tokens actually changed is around five times larger than the count of tokens predicted to be changed for our subject systems. Thus, in-spite of underestimation we realize that *it is possible to predict the number to tokens that might need to be changed for changing a particular method using our effort estimation tool.*

TABLE IV. Correlation between token counts

	Ant-Contrib	Carol	Ctags	Qmail Admin	Greenshot	Capital Resource
CACT	154.51	152.18	101.67	557.55	277.82	30.87
CTPC	31.33	18.1	21.84	100.93	45.66	40.23

Pearson Correlation Coefficient between CACT and CTPC = 0.888754

$CACT$ = Count of Actually Changed Tokens

$CTPC$ = Count of Tokens Predicted to be Changed

VII. Threats to Validity

We conduct our investigations by detecting code clones using NiCad [8]. While all clone detectors suffer from the *confounding configuration choice problem* [31] and might provide different clone detection results for different settings, the settings that we have used for NiCad are considered standard [27]. NiCad can detect code clones with high precision and recall with these settings [27], [28]. Thus, we believe that our findings in this experiment are important.

Our experimental results are based on only six open source subject systems which is not sufficient for taking any concrete decision. However, our selected systems are varying in size, application domains, revisions and programming languages. We selected the systems in this way intentionally to avoid possible bias of system size, domain and programming language on effort. Thus we believe that our investigation has brought out some important insights regarding the maintenance efforts for cloned and non-cloned code.

VIII. Related Work

Hotta et al. [10] studied the impact of clones by measuring the modification frequencies of the duplicated and non-duplicated code segments. They conducted a fairly large study using different tools and subject systems which suggests that the presence of clones does not introduce extra difficulties to the maintenance phase. Krinke [15] investigated Java, C and C++ code bases considering Type-I clones, and analyzed how consistently code clones are changed during maintenance. He found that clone groups changed consistently through half of their lifetime. In other two experiments [16], [17] he showed that cloned code is more stable than non-cloned code. Lozano and Wermelinger [21]–[23] conducted a number of experiments using CCFinder [12] to assess the effects of clones on the change-proneness of software systems. They found that code clones can often exhibit higher change-proneness than non-cloned code. Juergens et al.'s [11] study with large scale commercial systems suggests that inconsistent changes are very frequent to the cloned code and nearly every second unintentional inconsistent change to a clone leads to a fault. Kapser and Godfrey [13] identified different patterns of cloning and experienced that around 71% of the clones could be considered to have a positive impact on the maintainability of the software system. Aversano et al. [2] combined clone detection and modification transactions on open source software repositories to investigate how clones are maintained during the evolution and bug fixing. Their study reports that most of the cloned code is consistently maintained. In another similar but extended study, Thummalapenta et al. [30] indicated that most of the cases clones are changed consistently and for the remaining inconsistently changed cases clones mainly undergo independent evolution.

We see that different studies have tried to investigate the impacts of clones in different ways, however, no studies have measured how much effort is spent for cloned and non-cloned portions of a code base. In this paper we investigate this issue and try to find a concrete answer to the question 'Does the presence of clones increase maintenance effort?'. Our findings establish that cloned code often requires higher maintenance effort compared to non-cloned code. We also compare the maintenance efforts required by three types (Type 1, Type 2, and Type 3) of code clones and find that Type 2 and Type 3 clones require higher efforts than Type 1 clones. None of the existing studies have compared different clone-types considering the maintenance efforts they require. Our findings are important for prioritizing code clones for management.

IX. Conclusion

In this paper, we present an empirical study to determine and compare the maintenance efforts required for cloned and non-cloned code. For the purpose of our study we implement a prototype tool which is capable of performing two tasks: (i) calculation of effort for changes that previously occurred in a particular method, and (ii) prediction of effort that might need to be spent for future changes to a particular method. Our experimental results on six candidate subject systems written in three different programming languages show that:

- Cloned code often requires a higher amount of change effort during maintenance compared to non-cloned code.
- Both Type 2 and Type 3 clones require more maintenance effort than Type 1 clones.

We also observe that Type 3 clones generally require a higher amount of modifications (in terms of tokens) compared to the other two clone-types (Type 1, and Type 2). According to our findings we decide that Type 2 and Type 3 clones should be given more care during development. These two types of code clones should be prioritized for management.

We also evaluate our tool's predictability and observe that it can be used to predict the number of tokens that might need to be changed for changing a particular method. Our effort estimation tool can be helpful from both managerial and development perspectives. Managers will be able to calculate developer efforts spent on source code modifications. Developers can estimate effort for changing any existing method and can decide whether to modify existing methods or to create new ones for incorporating some changes to a particular project. As a future work, we are planning to integrate our tool with Eclipse IDE as a plug in.

Acknowledgments: This work is supported in part by the Natural Science and Engineering Research Council of Canada (NSERC).

References

[1] Y. Ahn, J. Suh, S. Kim, H. Kim, "The software maintenance project effort estimation model based on function points", *Journal of Software Maintenance and Evolution: Research and Practice*, 2003, 15(2):71-85.

[2] L. Aversano, L. Cerulo, M. D. Penta, "How clones are maintained: An empirical study," Proc. *CSMR*, 2007, pp. 81-90.

[3] L. Barbour, F. Khomh, Y. Zou, "Late Propagation in Software Clones", Proc. *ICSM*, 2011, pp. 273 – 282.

[4] L. Barbour, F. Khomh, Y. Zou, "An empirical study of faults in late propagation clone genealogies", *Journal of Software: Evolution and Process*, 2013, 25(11):1139 – 1165.

[5] B. W. Boehm, R. Madachy, and B. Steece, *Software Cost Estimation with Cocomo II.* Upper Saddle River, NJ: Prentice Hall, 2000.

[6] S. Bouktif, G. Antoniol, E. Merlo, M. Neteler"A Novel Approach to Optimize Clone Refactoring Activity", *GECCO*, 2006, pp. 1885 - 1892.

[7] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," Proc. *11th IEEE International Software Metrics Symposium*, 2005, p. 29 – 37.

[8] J. R. Cordy, C. K. Roy, "The NiCad Clone Detector", Proc. *ICPC Tool Demo*, 2011, pp. 219 – 220.

[9] N. Göde, Rainer Koschke, "Frequency and risks of changes to clones", Proc. *ICSE*, 2011, pp. 311 – 320.

[10] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto, "Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software," Proc. *EVOL/IWPSE*, 2010, pp. 73–82.

[11] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, "Do Code Clones Matter?," Proc. *ICSE*, 2009, pp. 485– 495.

[12] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," TSE, 28(7), 2002, pp. 654–670.

[13] C. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software," Emp. Soft. Eng. 13(6), 2008, pp. 645–692.

[14] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," Proc. *ESEC-FSE*, 2005, pp. 187–196.

[15] J. Krinke, "A study of consistent and inconsistent changes to code clones," Proc. *WCRE*, 2007, pp. 170–178.

[16] J. Krinke, "Is cloned code more stable than non-cloned code?," Proc. *SCAM*, 2008, pp. 57–66.

[17] J. Krinke, "Is Cloned Code older than Non-Cloned Code?," Proc. *IWSC*, 2011, pp. 28–33.

[18] Y. Ku, J. Du, Y. Yang, Q. Wang, "Estimating Software Maintenance Effort from Use Cases: an Industrial Case Study", in Proc. *ICSM*, 2011, pp. 482 - 491

[19] H. Leung, "Estimating maintenance effort by analogy," *Empirical Software Engineering*, vol. 7, 2002, pp. 157-175.

[20] J. Li, M. D. Ernst, "CBCD: Cloned Buggy Code Detector", Proc. *ICSE*, 2012, pp. 310 – 320.

[21] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the Harmfulness of Cloning: A Change Based Experiment," Proc. *MSR*, 2007, pp. 18–21.

[22] A. Lozano and M. Wermelinger, "Tracking clones imprint," Proc. *IWSC*, 2010, pp. 65–72.

[23] A. Lozano, and M. Wermelinger, "Assessing the effect of clones on changeability," Proc. em ICSM, 2008, pp. 227–236.

[24] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, K. A. Schneider, "Comparative Stability of Cloned and Non-cloned Code: An Empirical Study", Proc. *SAC*, 2012, pp. 1227 – 1234.

[25] M. Mondal, C. K. Roy, K. A. Schneider, "An Empirical Study on Clone Stability", *SIGAPP Applied Computing Review*, 2012, 12(3): 20 – 36.

[26] M. Mondal, C. K. Roy, K. A. Schneider, Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems, Proc. *CASCON*, 2012, pp. 205 – 219.

[27] C.K. Roy and J.R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," Proc *ICPC*, 2008, pp. 172–181.

[28] C. K. Roy and J. R. Cordy, "A mutation / injection-based automatic framework for evaluating code clone detection tools," Proc. *Mutation*, 2009, pp. 157–166.

[29] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating code clone genealogies at release level: An empirical study," Proc. *SCAM*, 2010, pp. 87–96.

[30] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta, "An empirical study on the maintenance of source code clones," *ESE*, 15(1), 2009, pp. 1–34.

[31] T. Wang, M. Harman, Y. Jia, J. Krinke, "Searching for Better Configurations: A Rigorous Approach to Clone Evaluation", Proc. *ESEC/SIGSOFT FSE*, 2013, pp. 455 – 465.