Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems

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

Software maintenance is an important and challenging phase of the software development life cycle because changes during this phase without proper awareness of dependencies among program modules can introduce faults in the software system. There is also a common intuition that cloned code introduces additional software maintenance challenges and difficulties. To support successful accomplishment of maintenance activities we consider two issues: (i) identifying coding characteristics that cause high source code modifications, and (ii) guidance for minimizing source code modifications.

Focusing on these two issues we investigated the effects of method sharing (among different functionality) on method co-changeability and source code modifications. We proposed and empirically evaluated two metrics, (i) *COMS* (Co-changeability of Methods), and (ii) *CCMS* (Connectivity of Co-changed Method Groups). *COMS* measures the extent to which a method co-changes with other methods. *CCMS* quantifies the extent to which a particular functionality in a software system is connected with other functionality in that system. In other words *CCMS* measures the intensity of method sharing among different functionality or tasks (defined later). We investigated the impact of *CCMS* on *COMS* and source code modifications. Our comprehensive study on hundreds of revisions of six open source subject systems covering three programming languages (Java, C and C#) suggests that - (i) higher *CCMS* causes higher *COMS* as well as increased source code modifications, (ii) *COMS* in the cloned regions of a software system is negligible as compared to the *COMS* in the non-cloned regions, and (iii) in-spite of some issues (described later) cloning can be a possible way to reduce *CCMS*.

1 Introduction

The software maintenance phase of the software development life cycle is undoubtedly important and challenging. According to some recent statistics [30], about 60 to 90 percent of annual software expenditures involves maintenance to existing software systems. The maintenance phase consists of those changes that are made to a software system after it has been deployed to the client upon client acceptance. The changes occurring in this phase are sometimes extremely challenging because changes without proper consideration of the impact and the underlying dependencies among related program modules can introduce faults in the software system. Low couplings or dependencies among program modules is always desirable [24]. A well designed software system always minimizes coupling [23].

Existing studies have focused on programmer awareness of dependencies among program artifacts. There are numerous studies on: (i) the detection of class, method or file level interdependencies and co-change patterns [5–7, 12, 31, 33], (ii) visualization of these dependencies and patterns [2–4], (iii) impacts of changing program components [7, 8], and (iv) propagation of changes [14] based on the software evolution history. However, none of these studies focused on the following two points.

(1) Minimization of changes: Each of these studies support programmer awareness about which other entities might need to be modified while modifying a particular entity. But, none of these studies investigated the reasons why some software systems exhibit higher changeability than others. None of these studies provide a way to reduce the number of modifications.

(2) Minimization of module dependencies: None of these studies provide a way to minimize dependencies (couplings) among program modules. Intuitively, higher dependency among program modules makes the modifications to the modules more difficult. There are many refactoring mechanisms but these cannot remove complex dependencies in many situations. Suppose a particular user-defined method is being used by two different functionalities (or tasks). While making changes to this method we should be concerned about all other methods implementing these two functionalities. In a real scenario, a particular method might be used by many functionalities or tasks (defined in Section 3) and in that case, modifications to the shared method will become more difficult. If we cannot eliminate such complex dependencies, the software may become increasingly complex over time, and may go beyond the point of maintainability. None of the existing studies shows possible ways of minimizing module dependencies.

Focusing on these two issues we performed an in-depth empirical study with the following contributions.

(1) Our study discovers a potential cause of increased modifications to program artifacts.

(2) We propose a possible way to minimize module dependencies (or couplings) as well as source code modifications.

We performed our investigation considering method level granularity. We introduce two metrics: **(1)** *COMS* (Co-changeability of Methods), and **(2)** *CCMS* (Connectivity of Co-changed Method Groups). ***COMS*** quantifies the extent to which a particular method co-changes with other methods. ***CCMS*** measures the sharing of methods among different functionalities or tasks. In other words, CCMS is a measure of dependencies (or couplings) among methods. In this paper, we extensively investigated the influence of **CCMS** on both **COMS** and source code modifications.

We performed a case study with hundreds of revisions of six open source software systems written in three different languages and evaluated the introduced metrics in three ways: **(i)** for the whole software system, **(ii)** for the cloned regions of the system, and **(iii)** for the non-cloned regions of the system. We also measured a change related metric ***CMP*** (Code Modification Probability) and observed whether higher connectivity among co-changed method groups causes increased source code modifications.

According to our experimental results, **(i)** *higher* ***CCMS*** *causes higher* ***COMS***, **(ii)** *higher* ***CCMS*** *is also a possible cause of increased source code modifications as well as efforts in the maintenance phase (empirically evaluated with statistical support)*, **(iii)** *the* ***COMS*** *in the cloned regions of a software system is negligible as compared to the* ***COMS*** *in the non-cloned regions* and **(iv)** *cloning can be a possible way of minimizing* ***CCMS*** *for those situations where functionalities or tasks are connected but are likely to evolve independently.*

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 explains some terminology including task, co-changed method group, COMS and CCMS. Section 4 describes the co-changeability of methods in cloned and non-cloned code. Section 5 discusses our methodology. Section 6 and Section 7 contain the experimental setup, results and analysis of results. Section 8 discusses the minimization of co-changeability and some possible validity threats are discussed in Section 9. Finally in Section 10, we conclude the paper mentioning our future work.

2 Related Work

Studying the impacts of co-changes is not a new topic. Jafar et al. [16] performed a comprehensive study on macro co-changes considering file level granularity. They introduced two metrics *MCC* (macro co-changes) and *DMCC* (diphase macro co-changes) and using their proposed approach *Macocha* they detected how many files exhibit *MCC* and *DMCC*. Their introduced metrics can assist in mainly two ways: (i) by managing development teams, and (ii) by managing bugs and change propagations.

Zimmermann et al. [33] implemented a tool called *ROSE* and integrated it with *ECLIPSE* as a plugin to achieve three aims: (i) prediction of future changes, (ii) determination of component (file, method, variable etc) couplings that are difficult to detect by program analysis, and (iii) prevention of errors because of incomplete changes. Their *ROSE* prototype could predict which files need to be modified for a particular change request for 26% of cases.

Beyer [3] implemented and described a co-change visualization tool *CCVISU* that can extract the underlying clustering of artifacts in a software system by analyzing CVS log files. *CCVISU* can help us in two ways: (i) understanding the relationships among different software artifacts such as files, classes, methods and packages which is useful for reverse engineering, and (ii) helpful guidance of changes happening in the maintenance phase.

Canfora and Cerulo [7] proposed an impact analysis approach that retrieves the set of affected source files from a change request by mining the information stored in the bug tracking and versioning systems.

Ying et al. [31] proposed and developed a method which was capable of recommending relevant source files for a particular modification task by querying previously stored change patterns. The change patterns were extracted by mining data from a software configuration management (SCM) system by applying an association rule mining algorithm. Their recommendation system is intended to reveal valuable dependencies among files.

Gall et al. [11] introduced an approach of discovering logical dependencies and change patterns among different program modules by using the information in the release history of a system. Their logical coupling identification approach is intended to be used to restructure systems to minimize structural problems.

D'Ambros et al. [2] implemented *evolution radar* to visualize module level and file level logical couplings. They argued that their tool integrated with a development environment can support restructuring, redocumentation and change impact estimation.

Hassan and Holt [14] conducted an empirical study on change propagations in the software systems. They have shown that historical co-change information can be used to help developers during the change propagation process.

Zhou et al. [32] presented a Bayesian network based approach for predicting change coupling behaviour between source code artifacts. On the basis of a set of extracted features such as: static source code dependency, frequency of previous co-changes, change significance level, age of change and co-change entities their approach models the uncertainty of change coupling process.

We can see that, none of the existing studies focused on how the connectivities among different functionalities affect co-changeability of methods and source code modifications. Our study investigates this issue.

3 Terminology

3.1 Task

Before the commencement of a particular project it is generally decomposed into multiple smaller tasks which are assigned to the programmer responsible for the task. If we consider an example project 'Library Management System', an example task could be 'User Login' or 'Issuing a Book' etc. While accomplishing a particular task a developer can further decompose it into multiple methods or classes or other language specific structures which together perform the particular task. In this paper we focus on the methods that belong to a particular task.

3.2 Co-changed Method Group

If a particular project is developed under version controlling system such as SVN, the responsible programmer commits the relevant source code files (possibly with other non-source-code files if necessary) to SVN after partial or full implementation of each task. So it is very likely that the methods which are added or modified in a particular commit operation belong to a particular task. According to our definition these methods which are added or modified in a particular commit operation form a co-changed method group.

During the evolution of a software system multiple revisions of it are created because of multiple commit operations. 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. As we have already mentioned, the methods that are created or that receive some changes in a particular commit operation generally belong to a particular task. But, a single commit operation might also affect multiple tasks. Such commits are termed 'atypical commits' [21]. For excluding atypical commits Lozano and Wermelinger [21] discarded 2.5% of the largest commit operations while analyzing the revisions of a subject system.

Detection of Co-changed Method Groups: To determine the co-changed method group for a particular commit operation $commit(i)$ we need to accomplish several tasks: (i) detection of all methods in $revision(i)$ with corresponding beginning and ending line numbers, (ii) determination of changes in $revision(i)$ with corresponding line numbers, (iii) mapping these changes to the detected methods of $revision(i)$ and at last (iv) retrieval of the changed methods. For a sequence of n revisions of a subject system we will get a sequence of $n-1$ commit operations. After discarding the commit operations with atypical changes we will obtain our target commit operations. If the count of these target commit operations is t, we will have t co-changed method groups. But, it is very likely that, a particular co-changed method group will appear multiple times in this sequence of t groups because, changes in multiple commit operations might be centered around the same or similar tasks. So, we need to determine the unique co-changed method groups for a sequence of commit operations. Unique co-changed group identification process is elaborated in *Algorithm* 1 (Section 5).

Difference Between Our Proposed Method and Existing Methods: Our process of detecting co-changed method groups is a variant of the association rules introduced by Zimmerman et al. [33]. The original association rule considers the co-changes of all types of entities including variables, methods, classes etc. In our implementation we have considered the co-changes among methods only. The method proposed by Zimmerman et al. [33] is not suitable for the investigation regarding method sharing. Because, it does not show how methods are grouped for accomplishing different functionalities. Also, no other existing methods tried to extract possible groups of co-changing entities. Our proposed algorithm (Algorithm 1) detects possible groups of co-changing methods. From these groups we can easily identify shared methods (the methods that are shared by more than one group) and can calculate **CCMS** and **COMS**.

Minimization of the probability of false associations: Generally while developing or making changes to a particular functionality a programmer writes code, checks whether the code is working properly and continues this process until they achieve the required goal. During this process of writing and checking the programmer might erroneously change program artifacts that are not related to the functionality. These errors can be fixed through a continuous checking and writing process. When the programmer commits the changes to SVN, they try to be sure that the changes they make do not contain bugs because the committed code is likely used by other programmers. So, we see that before committing the programmer remains more concerned about the accuracy of the code. So, if we associate the entities changed in a commit operation, it is likely that we will avoid irrelevant or false associations.

Also, discarding atypical commits increases the probability of avoiding false associations.

3.3 Co-changeability of Methods (COMS)

Consider that we detected m unique co-changed method groups. Each of these groups is denoted by $g(i)$ where $1 \leq i \leq m$. The count of elements in a particular group $g(i)$ is denoted by $|g(i)|$. If we want to change a particular method in a group $g(i)$, we also need to be concerned about the other $|g(i)| - 1$ members belonging to this group. So, we calculate the COMS for group $g(i)$ according to the Eq. 1.

$$COMS(g(i)) = |g(i)| \times (|g(i)| - 1) \quad (1)$$

Here, $COMS(g(i))$ is the COMS for group $g(i)$. The total COMS of all groups in a software system can be calculated using the following equation.

$$Total\ COMS = \sum_{i=1}^{m} COMS(g(i)) \quad (2)$$

Here m is the total number of unique co-changed method groups in the software system. Co-changeability is influenced by group connectivity and this influence is not trivial. In the following subsections we at first define CCMS and then mathematically show the influence of CCMS on COMS.

3.4 Connectivity of Co-changed Method Groups (CCMS)

If two co-changed method groups share a common subset of methods, we say that these two groups are connected. We define the connectivity of this connection constructed by these two co-changed method groups by the count of methods shared between these two groups. A particular co-changed method group can have multiple connections with multiple other groups.

Suppose a group $g(i)$ has n connections. The subsets of methods corresponding to a connection is denoted by $s(j)$ where $1 \leq j \leq n$. The connectivity of this group $g(i)$ can be calculated by the summation of the connectivities of these connections in the following way.

$$CCMS(g(i)) = \sum_{j=1}^{n} |s(j)| \quad (3)$$

Here, $CCMS(g(i))$ is the connectivity of the co-changed method group g(i).

We should note that, this equation (Eq. 3) gives more emphasis on those methods that are included in higher number of connections. Suppose, a particular method group has n connections with n other groups. If a particular method in this group remains included in m of these connections where $m \leq n$, this method will be considered m times by this equation.

3.5 Influence of CCMS on COMS

Suppose, there are m methods which are divided into two co-changed groups where each group has r methods. Also, these groups share a common subset of methods. If the common subset contains c methods, the connectivity of the connection between these two groups $= c$ (CCMS $= c$). Now the total co-changeability of the methods of these two groups can be calculated in the following way.

$$Total\ COMS(TCOMS) = 2 \times r \times (r - 1) \quad (4)$$

But, $m = 2 \times r - c$ (because each of the two groups has r methods and c methods are common for each group). So, $r = (m + c)/2$. Using this value of r in *Eq.* 3, we get,

$$TCOMS = 2 \times \frac{m+c}{2} \times (\frac{m+c}{2} - 1)$$
$$= \frac{(m+c) \times (m+c-2)}{2} \quad (5)$$

Examining the above equation we can state that, *greater values of c will cause greater COMS*. However, in real scenario, a single group might have several connections with several other groups which will increase the probability of method co-changes to a great extent. So, *by reducing CCMS we can reduce COMS.*

4 COMS in Cloned and Non-cloned Code

We derived the equation for calculating COMS for the whole software system from its observed

unique co-changed method groups. We can also calculate COMS separately for cloned and non-cloned regions of a software system. We have done this because there are many empirical studies [13,15,17–21] with controversial outcomes about the impacts of clones in the maintenance phase. Some studies [17, 20, 21] imply that that cloned code is more harmful than non-cloned code while others [13, 15, 18, 19] report the opposite. We wanted to find whether cloned code exhibits more co-changeability of methods than non-cloned code or not.

We have already discussed the procedure for calculating the co-changed method group for a particular commit operation *commit*(i) applied on a revision *revision*(i). If we apply a clone detection tool on *revision*(i), we can separate the cloned and non-cloned blocks belonging to *revision*(i). By mapping these blocks to the methods of this revision we can determine which methods are cloned and which methods are not. We also need to know which lines of a particular method are cloned. According to our consideration, a cloned method might be fully or partially cloned. A partially cloned method has some non-cloned blocks in it. As we know the cloned lines of each method, we can map the changes belonging to commit operation *commit*(i) to the cloned and non-cloned portions of the methods. Thus, we can determine the group of methods which have changes in their cloned portions and also the group of methods which have changes in their non-cloned portions. We call these groups co-changed method groups of cloned and non-cloned code respectively. A partially cloned method which has some changes in its non-cloned portions but not in its cloned portions will belong to the co-changed method group of non-cloned code not to the group of cloned code. By determining the unique co-changed method groups in cloned and non-cloned regions we can determine the COMS for cloned and non-cloned regions.

5 Methodology

The calculation of unique co-changed method groups, their connectivities and co-changeabilities for a particular software system requires several sequential steps.

5.1 Preprocessing

We examined all the revisions (from revision 1 to the last revision as indicated in Table 2) of a software system for calculating metrics. We know that a particular revision is created because of a particular commit operation. A commit operation might (or might not) consist of some source code changes. We considered only those commit operations that consist of some source code changes. We extracted the revisions corresponding to these commit operations. In other words, we worked only on those revisions which were created because of some code changes. Two pre-processing steps have been applied to each of the target revisions of a system: **(i)** deletion of lines containing only a single brace ('{' or '}') and appending the brace at the end of previous line; and, **(ii)** removal of comments and blank lines from source files.

5.2 Method detection and extraction

For detecting the methods from the source files of a specific revision we used CTAGS [10]. For each method we collected its: (i) file name, (ii) class name (Java and C# systems), (iii) package name (Java), (iv) method name, (v) signature, (vi) starting line number and (vii) ending line number. We also assigned a unique ID to each method. However, the ID of a method of one revision can be the same as that of a method of another revision. This does not introduce conflicts because a separate file is generated for each revision. Starting and ending line numbers of methods are necessary during the mapping of clones and changes to the methods. Method extraction can be accelerated by reusing the results from the previous revision. Suppose we already detected the methods of revision *revision*(i). For the very next revision *revision*($i+1$) we do not need to extract methods from all the files. We should extract methods only from those files which have changed. For the unchanged files, methods can be collected from the results stored for the previous revision. This mechanism can be followed for the other subsequent revisions.

We do not store the methods at this moment. Before storing we also need to know which parts

of a method are cloned and which parts are not cloned and whether some cloned or non-cloned parts have changed before being forwarded to the next revision.

5.3 Clone detection

We have used the recently introduced hybrid clone detection tool NiCad [26] that combines the strengths and overcomes the limitations of both text-based and AST based clone detection techniques and exploits novel applications of a source transformation system to yield highly accurate identification of Type-1, Type-2 and Type-3 clones in software systems [27, 28].

5.4 Detection and reflection of changes

We identified the changes between corresponding files of consecutive revisions using UNIX *diff* command. *diff* outputs three types of changes: (i) addition, (ii) deletion, and (iii) modification with corresponding line numbers. We mapped these changes to methods using line information to determine which methods have changed in a particular commit operation.

5.5 Storage of methods

At this stage, we have all the necessary information of all methods belonging to a particular revision. We store these methods in an XML file with individual entry for each method. For each revision we generated a separate XML file containing the methods of the corresponding revision. A file name is constructed by appending the revision number at its end so that, we can generate it when necessary (for getting previously stored methods of the unchanged files of a former revision).

5.6 Method genealogy detection

For mapping methods between two consecutive revisions we followed the origin analysis technique proposed by Lozano and Wermelinger [21]. This technique uses a combination of location and signature similarities to determine which method of $revision(i)$ corresponds to which method of $revision(i+1)$. Some methods of $revision(i)$ might get deleted in $revision(i+1)$ and also some new methods might be created in $revision(i+1)$. We stored the mapping information for each two consecutive revisions in a separate file. Method mapping was accomplished using method IDs. The file names contain the revision numbers in a disciplined way so that we can generate them when necessary.

5.7 Determination of unique co-changed method groups

As we are inspecting the methods affected by the commit operations sequentially, we are storing and updating the co-changed method groups according to the algorithm *Algorithm* 1. For eliminating the effects of atypical changes [21] that affect multiple functionalities (or goals) at a single commit operation we discarded 2.5% of the largest commit operations for each of the subject systems from our consideration as was done by Lozano and Wermelinger [21]. From each of the commit operations of our target set obtained by eliminating the commit operations with atypical changes, we extracted the co-changed method groups sequentially. Method genealogy extraction was necessary to determine whether a currently detected group has already appeared previously.

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 at 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 beginning of this process (while examining the first commit operation), the *existing list* remains empty. These sequential steps are elaborated in the *Algorithm* 1.

Though we have discarded 2.5% of the largest commit operations (atypical commits), there is little probability that a co-changing method group will contain unrelated methods.

However, the algorithm ensures the detection of all possible groups.

Algorithm 1 Determine unique co-changed method groups

Require: The sequence of commit operations, *ExistingList* of groups (initially empty)
Ensure: Unique co-changed method groups.
  for each commit operation *commit*(i) **do**
    $NewGroup \leftarrow$ the list of changed methods
    for each group $g(j)$ in *ExistingList* **do**
      if $NewGroup \subset g(j)$ **then**
        Ignore *NewGroup*
        exit the loop.
      else
        if $g(i) \subset NewGroup$ **then**
          Delete $g(i)$ from *ExistingList*
        end if
      end if
    end for
    if *NewGroup* is not ignored **then**
      Add *NewGroup* to the *ExistingList*.
    end if
  end for

5.8 Metric Calculations

5.8.1 Calculation of CCMS and COMS

After determining the unique co-changed method groups we calculated the values of our proposed metrics, CCMS and COMS, for all method groups in total according to the processes described in Section 3.3 and Section 3.4. However, while finding correlations, we calculated the average values per co-change method group for these metrics.

5.8.2 Calculation of CMP (code modification probability)

For finding correlations of code modifications with connectivity, we calculated the code modification probabilities (CMP) for each of the candidate software systems using the following equation.

$$CMP = \frac{\sum_{c \epsilon CMS} CML(c)}{|CMS| \times \sum_{c \epsilon CMS} LOC(c)} \quad (6)$$

In the above equation (Eq. 6), *CMP* is the code modification probability. $CML(c)$ denotes the count of modified (added, deleted or changed) lines of a software system during the commit operation c. *CMS* is the set of all commits where there were some modifications to the source code. $LOC(c)$ is the count of total lines of code of a candidate system during commit operation c.

We see that, Eq. 6 calculates the source code modification probability by considering only those commit operations where there were some modifications to the source code. For each of the commit operations with some modifications to the source code we calculated the following two measurements.

(1) Total number of lines in that revision on which the commit was applied ($LOC(c)$).

(2) Total number of lines modified because of the commit operation ($CML(c)$).

We know that each commit operation creates a new revision. For calculating the number of lines modified in a particular commit operations we at first identify two revisions: (1) the revision on which the commit operation was applied and (2) the revision that was created just after applying the commit operation. Then, we use UNIX *diff* to identify the lines that were modified in the older revision to create the newer one.

6 Experimental Setup

6.1 Implementation framework

We implemented our method in Java programming language using the Actor Architecture platform [1] which provides us a parallel and distributed framework suitable for coarse grained concurrency.

As we have already described in the previous section, the extraction of method genealogies is a huge task. This task can be divided into multiple smaller but similar tasks. These smaller tasks can be executed by different processes in parallel on different processors of the same machine or on different machines connected through an interconnection network. The results of these tasks can be combined to achieve the final result. The Actor Architecture framework helped us perform parallel and faster ex-

traction of method genealogies in the following way.

We visualized our parallel framework as a manager-worker paradigm where a single manager divides and distributes tasks among different workers working on different machines. At the very beginning of execution, the manager divides the whole range of revisions into a number of sub-ranges of equal length. Each sub-range contains multiple consecutive revisions and the count of subranges is equal to the number of workers. The manager then assigns each of the sub-ranges to a particular worker. Each worker is responsible for the extraction and mapping of the methods that were created in the revisions it has been assigned to. To get the final method mapping for the whole range of revisions, the workers need to synchronize among themselves. The synchronization among the workers and the manager is done by asynchronous message passing among them.

Here, we should mention that each of the workers executes the same instruction sequence. For implementing the manager and worker processes we extended the *Actor* class defined in the Actor Architecture platform.

6.2 Setup for NiCad

We used the NiCad [26] clone detection tool to detect clones in the subject systems. NiCad can detect both exact and near-miss clones at the function or block level of granularity. We detected block clones with a minimum size of 5 LOC. We used the NiCad settings in Table 1 for detecting three types of clones. The dissimilarity threshold means that the clone fragments in a particular clone class may have dissimilarities up to that particular threshold value. We set the dissimilarity threshold to 20% with blind renaming of identifiers for detecting Type-3 clones. For all these settings NiCad is shown to have high precision and recall [28]. Before using the NiCad outputs of Type-2 and Type-3 cases, we processed them in the following way.

(1) Every Type-2 clone class that exactly matched any Type-1 clone class was excluded from Type-2 outputs.

(2) Every Type-3 clone class that exactly matched any Type-1 or Type-2 class was excluded from Type-3 outputs.

Table 1: NiCad Settings for three types of clones

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

6.3 Subject Systems

Table 2 lists the subject systems included in our study along with their associated attributes. We downloaded these from open source SVN repositories. The subject systems are diverse, differing in size, spanning five different application domains, and covering three programming languages.

7 Experimental Results

We applied our methodology to six open source software systems and calculated the following for each of them.

(i) The count of unique co-changed method groups

(ii) Total COMS for all co-changed method groups

(iii) Total CCMS of all co-changed method groups

(iv) Code modification probability (CMP).

We provide these values in Tables 3 and 4.

We determined the Pearson correlations between CCMS and the other 2 measures in the above list (excluding the count of unique co-changed method groups) and recorded the results in Table 5. For calculating the correlations, we calculated the average values of CCMS and COMS per method group for each subject system. We also calculated the COMS and CCMS separately for cloned and non-cloned code for each of the subject systems.

Table 2: Subject Systems

	Systems	Domains	LOC	Revisions
Java	Carol	Game	25,092	1000
	Dnsjava	DNS protocol	23,373	1635
C	Ctags	Code Def. Generator	33,270	774
	Camellia	Multimedia	85,015	207
C#	GreenShot	Multimedia	37,628	999
	MonoOSC	Formats and Protocols	18,991	355

Table 3: Total CCMS and Total COMS for different subject systems

Language	Subject Systems	UGC	COMS	CCMS
Java	Carol	157	1578	118
	Dnsjava	329	4010	956
C	Ctags	142	1532	920
	Camellia	67	938	526
C#	GreenShot	241	3262	1741
	MonoOSC	105	2246	1785

UGC = Unique co-changed Group Count

Table 4: Source code modification probabilities for different subject systems

Language	Subject Systems	NCMS	CMP
Java	Carol	383	3.23E-06
	Dnsjava	1254	1.75E-06
C	Ctags	447	3.63E-06
	Camellia	147	5.24E-06
C#	GreenShot	586	2.99E-06
	MonoOSC	236	2.35E-05

CMP = Code modification probability

NCMS = Number of commits with modifications in the source code

Table 5: Correlation of CCMS with COMS and CMP

Lang.	Systems	CCMS	COMS	CCMS	CMP
Java	Carol	0.7515	10.0509	0.7515	3.23E-06
	Dnsjava	2.9057	12.1884	2.9057	1.75E-06
C	Ctags	6.4788	10.7887	6.4788	3.63E-06
	Camellia	7.8507	14	7.8507	5.24E-06
C#	GreenShot	7.22	13.5352	7.22	2.99E-06
	MonoOSC	17	21.3904	17	2.35E-05
Correlation Co-efficient		**0.941557804**		**0.901590668**	

CCMS = Connectivity of Co-changing method groups

COMS = Co-changeability of methods

CMP = Code modification probability

7.1 Analysis of Experimental Results

We determined the strengths of correlations [29] between CCMS and two other measures: COMS and CMP. The correlations recorded in Table 5 are explained below.

7.1.1 Correlation between CCMS and COMS

From Table 5 we see that, there is a strong correlation between connectivity and co-changeability. The Pearson correlation co-efficient between these two is 0.9415. Such a strong correlation is expected. Also, we have mathematically shown that, connectivity has direct influence on the co-changeability. Thus we can say that, *co-changeability can be minimized by minimizing connectivity.*

7.1.2 Correlation of CCMS with CMP

To determine whether higher connectivity is an indicator of higher modification of source code, we calculated the correlation between *CMP* (code modification probability) and *CCMS*. We observed that *CMP* is strongly correlated with connectivity with a correlation co-efficient of 0.9015. So, higher changeability (modification probability) in source code is an indicator of higher connectivity among co-changed method groups. As *CCMS* measures the intensity of method sharing among co-changing method groups, we can say that *higher method sharing might be a possible cause to higher changeability in source code.*

7.2 Clone related analysis

We calculated the COMS for non-cloned code and three types (Type 1, Type 2 and Type 3) of cloned code of a subject system separately to compare the COMS in cloned and non-cloned code. The average COMS of different types of cloned code and corresponding non-cloned code have been plotted in the graph in Fig. 1. In general we see that the COMS of each type of cloned code (except Type 1 clones of Dnsjava) is much smaller than non-cloned code. Also, we have observed that the numbers of co-changed method groups that we found for different types of cloned code are negligible compared to the group counts of non-cloned code. From this we decide that the COMS in cloned code is much lower than the COMS in non-cloned code. Clones are generally created for serving different tasks or functionalities independently, which indicates that cloned methods should have no coupling. This is obviously a good characteristic of clones, which can be carefully used to minimize the COMS in non-cloned code.

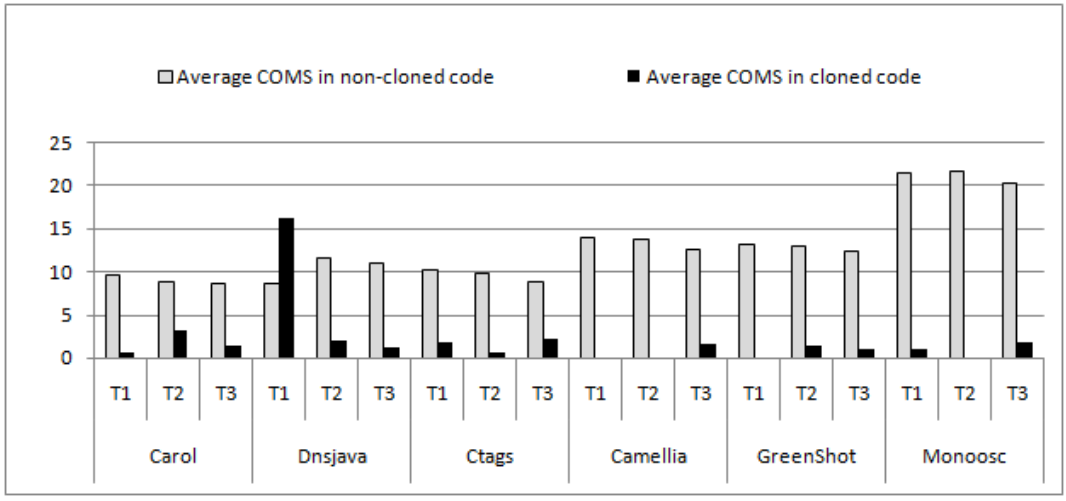

Figure 1: Comparison of COMS in non-cloned code and three types of cloned code.

8 Minimization of COMS

From our mathematical derivation in Section 3 we see that a higher CCMS (Connectivity of Co-changed Method Groups) causes a higher COMS (co-changeability of methods). Intuitively, higher co-changeability of methods should cause increased modifications to the source code. Also, in the analysis part we have seen that higher changeability in source code is an indicator of higher CCMS. COMS can be minimized by minimizing CCMS. There are two ways of minimizing CCMS. These are explained below.

8.1 Cloning

Minimization of CCMS is tricky. Cloning is a possible way of minimizing CCMS. If a particular method takes part in implementing several tasks, we can make a separate copy of this method for each of these tasks or functionalities. This is a way of minimizing CCMS without increasing group size. In this way, separate copies of this method will evolve independently with the evolutions of separate tasks. But for this to be a fruitful approach we need to be sure of the following.

(1) The method that is being replicated does not contain a bug. Identification of a bug in any of these copies will require the propagation of bug correction activities to all copies, which will likely increase change efforts.

(2) Cloning can be applied to minimize CCMS only for those situations where the tasks or functionalities that share the common method (or common set of methods) are likely to evolve independently. Otherwise, the synchronization of modifications among the cloned methods will increase the changeability of the source code as well as the maintenance effort.

In Table 6 we provide a simple example of two connected functionalities in QmailAdmin written in C. Each goal consists of methods from multiple files. The method 'int show_autorespond_line(char*, char*, time_t, char*)' of file 'autorespond.c' connects these two functionalities. This is a very simple connection, but when tasks (or functionalities) become in this way connected we can eliminate connectivity by cloning (assuming the shared method needs to evolve independently for each goal). Just for explanation we can say that, a separate copy of this method can be created so that two copies can serve two purposes (which might not be necessary for this case because of the simple connection between the two functionalities).

Table 6: Example of connected functionalities

Revision 140. Goal: Addition of new administrator	
File path	**Method signature**
qmailadmin/auth.c	set_admin_type()
qmailadmin/autorespond.c	int show_autorespond_line (char*,char*,time_t,char*)
qmailadmin/user.c	int addusernow()
Revision 144. Goal: Mail forwarding	
File path	**Method signature**
qmailadmin/alias.c	show_dotqmail_lines (char*,char*,time_t,char*)
qmailadmin/autorespond.c	int show_autorespond_line (char*,char*,time_t,char*)
qmailadmin/command.c	process_commands()
qmailadmin/forward.c	int show_forwards (char*,char*,time_t,char*)
qmailadmin/qmailadmin.c	main(argc,argv)
qmailadmin/util.c	int count_stuff(void)

8.2 Method breakdown

If a single method contains multiple sections for multiple functionalities (which is a common case for the C# systems according to our observation), we can split this method into several relevant methods containing the sections corresponding to those functionalities. In this way, we minimize method couplings as well as ensure securities for different functionalities such that, while changing the portion of a particular functionality other portions of the other functionalities will not be affected mistakenly.

9 Threats to Validity

The sample size of our study is not sufficient enough to draw a general conclusion. Furthermore, the level of expertise of the involved programmers and the nature of applications might also have some effects on the experimental results. However, our selection of six different subject systems of three different programming languages considering the diverse variety of sizes, application domains and hundreds of revisions should minimize these drawbacks considerably.

According to our definition and calculation procedure of co-changed method groups there is very little probability that a co-changed method group will contain unrelated methods and that co-changeability will be overestimated. Since we are identifying and updating the co-changed method groups from the very beginning of the development phase, we can retrieve all the existing connectivities as well as method sharing among different functionalities. Thus, we believe that, we could determine the actual effect of connectivities on co-changeabilities.

10 Conclusion

In this paper, we described an in-depth investigation on how the sharing of methods among different functionalities affects method co-changeability and modification of source code. For this purpose we proposed and empirically evaluated two metrics: (i) COMS (Co-changeability of Methods) and (ii) CCMS (Connectivity of Co-changed Method groups). The first metric measures the extent to which a particular method co-changes with other methods while the other one quantifies the sharing of methods among functionalities. We analyzed the influence of CCMS on both COMS and modifications of source code. According to our analysis, CCMS causes higher COMS as well as increased modifications in the source code.

We observed that, COMS in cloned code is

negligible compared to that of non-cloned code and also, cloning can be a solution to minimizing method group connectivity as well as couplings (or dependencies) among methods. Although our study is not exhaustive it contributes in two ways: **(1)** it identifies a possible cause of higher source code modifications and **(2)** it suggests a possible way of minimizing method couplings and code modifications. Our suggested solution has the potential to increase the stability of software systems in the maintenance phase. As future work we are planning to integrate our proposed methodology with the Eclipse IDE as a plugin so that we will be able to visualize the functionalities and their connectivities to find the target functionalities to minimize their connectivities.

About the Authors

Manishankar Mondal: Manishankar Mondal is an M.Sc student in the Department of Computer Science of the University of Saskatchewan. He is a lecturer (on leave) at Khulna University. His research interests are software maintenance and evolution.

Chanchal K. Roy: Chanchal Roy is an assistant professor of Software Engineering/Computer Science at the University of Saskatchewan. His research interests are in the area of software engineering. In particular, he is interested in software maintenance and evolution, including clone detection and analysis, reverse engineering, empirical software engineering and mining software repositories.

Kevin A. Schneider: Kevin Schneider is a Professor of Computer Science, Special Advisor ICT Research and Director of the Software Engineering Lab at the University of Saskatchewan. Dr. Schneider has previously been Department Head (Computer Science), Vice-Dean (Science) and Acting Chief Information Officer and Associate Vice-President Information and Communications Technology.

Before joining the University of Saskatchewan, Dr. Schneider was CEO and President of Legasys Corp., a software research and development company specializing in design recovery and automated software engineering. His research investigates models, notations and techniques that are designed to assist software project teams develop and evolve large, interactive and usable systems. He is particularly interested in approaches that encourage team creativity and collaboration.

References

[1] Actor Architecture platform. `http://www.docstoc.com/docs/5693312/Actor-Architecture`

[2] M. D'Ambros, M. Lanza, M. Lungu, "Visualizing co-change information with the evolution radar," *TSE*, 2009, vol. 35, no. 5, pp. 720–735.

[3] D. Beyer. "Co-change visualization", Proc. *ICSM*, 2005, Industrial and Tool volume, pp. 89–92

[4] D. Beyer, A. E. Hassan, "Animated visualization of software history using evolution storyboards," Proc. *WCRE*, 2006, pp. 199–210.

[5] S. Bouktif, Y.-G. Gueheneuc, G. Antoniol, "Extracting changepatterns from cvs repositories," Proc. *WCRE*, 2006, pp. 221–230.

[6] G. Canfora, M. Ceccarelli, L. Cerulo, M. Di Penta, "Using multivariate time series and association rules to detect logical change coupling: An empirical study," Proc. *ICSM*, pp. 1–10.

[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.

[8] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta, "An eclectic approach for change impact analysis," Proc. *ICSE*, 2010, pp. 163–166.

[9] J. Cordy, The txl source transformation language, *Sci. of Com. Prog.*, 61(3):190–210, 2006.

[10] Exuberant CTAGS: `http://ctags.sourceforge.net/`

[11] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," Proc. *ICSM*, 1998, pp. 190–199.

[12] D. M. German, "An empirical study of fine-grained software modifications," *Empirical Softw. Engg. Kluwer Academic Publishers*, September 2006, vol. 11.

[13] N. Göde, J. Harder, "Clone Stability", Proc. *CSMR*, 2011, pp. 65-74.

[14] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," Proc. *ICSM*, 2004, pp. 284–293.

[15] 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.

[16] F. Jaafar, Y. Gueheneuc, S. Hamel, G. Antoniol, "An Exploratory Study of Macro Co-changes", Proc. *WCRE*, 2011, pp. 32–334.

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

[18] 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.

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

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

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

[22] 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.

[23] A. J. Offutt, M. J. Harrold, P. Kolte, "A software metric system for module coupling", Proc. *Journal of Systems and Software*, 1993, pp. 295–308

[24] M. Page-Jones, "The practical guide to structured systems design", YOURDON Press, New York, NY, 1980.

[25] B. Pugsley, "Econ 210: Linear Regression Review", May 25, 2008. Available at: `http://home.uchicago.edu/~bpugsley/notes/regression_study_guide.pdf`

[26] 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.

[27] C.K. Roy and J.R. Cordy, "Near-miss Function Clones in Open Source Software: An Empirical Study," *Journal of Soft. Maintenance and Evolution: Research and Practice*, 22:3, pp. 165-189.

[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] Significance of the Correlation Coefficient. `http://janda.org/c10/Lectures/topic06/L24-significanceR.htm`

[30] Software Maintenance Costs. `http://users.jyu.fi/~koskinen/smcosts.htm`

[31] A. T. T. Ying, G. C. Murphy, R. Ng, M. C. Chu-Carroll, "Predicting source code changes by mining change history," *TSE*, 2004, vol. 30, no. 9, pp. 574–586.

[32] Y. Zhou, M. Wursch, E. Giger, H. C. Gall, and J. Lu, "A bayesian network based approach for change coupling prediction," Proc. *WCRE*, 2008, pp. 27–36.

[33] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, "Mining version histories to guide software changes," Proc. *ICSE*, 2004, pp. 563–572.