Science of Computer Programming 95 (2014) 445–468

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study

Manishankar Mondal *, Chanchal K. Roy, Kevin A. Schneider

University of Saskatchewan, Canada

ARTICLE INFO

Article history:
Received 11 December 2012
Received in revised form 6 November 2013
Accepted 12 November 2013
Available online 18 November 2013

Keywords:
Dispersion
Instability
Clones
Software maintenance
Code change

ABSTRACT

In this paper, we present an in-depth empirical study of a new metric, *change dispersion*, that measures the extent changes are scattered throughout the code of a software system. Intuitively, highly dispersed changes, the changes that are scattered throughout many software entities (such as files, classes, methods, and variables), should require more maintenance effort than the changes that only affect a few entities. In our research we investigate change dispersion on the code-base of a number of subject systems as a whole, and separately on each system's cloned and non-cloned code. Our central objective is to determine whether cloned code negatively affects software evolution and maintenance. The granularity of our focus is at the method level.

Our experimental results on 16 open source subject systems written in four different programming languages (Java, C, C#, and Python) involving two clone detection tools (CCFinderX and NiCad) and considering three major types of clones (Type 1: exact, Type 2: dissimilar naming, and Type 3: some dissimilar code) suggests that change dispersion has a positive and statistically significant correlation with the change-proneness (or instability) of source code. Cloned code, especially in Java and C systems, often exhibits a higher change dispersion than non-cloned code. Also, changes to Type 3 clones are more dispersed compared to changes to Type 1 and Type 2 clones. According to our analysis, a primary cause of high change dispersion in cloned code is that clones from the same clone class often require corresponding changes to ensure they remain consistent.

1. Introduction

Code cloning is a common yet controversial practice that studies have shown to have both positive [1,5,9,11,14,15,7] and negative [10,16–18] implications during software development and maintenance. Reuse of code fragments with or without modifications by copying and pasting from one location to another is very common in software development. This results in the existence of the same or similar code blocks in different components of a software system. Code fragments that are exactly the same or are very similar to each other are known as clones. Three types of clones are commonly studied: Type-1 (exact clones), Type-2 (clones with dissimilar naming), and Type-3 (clones with dissimilar naming and/or with some dissimilar code). The impact of clones on software maintenance is of great interest. Researchers have investigated the stability of cloned and non-cloned code [5,13–17,22,19,21,6] using a number of approaches in order to quantify clone impact. The idea is that if cloned code is less stable (e.g., numerous changes) than non-cloned code during maintenance, it is an indication that clones require more maintenance effort than non-cloned code [5]. If this is the case, clones may be considered harmful in the maintenance phase. However, existing approaches for measuring stability are insufficient.

* Corresponding author.
E-mail addresses: mshankar.mondal@usask.ca (M. Mondal), chanchal.roy@usask.ca (C.K. Roy), kevin.schneider@usask.ca (K.A. Schneider).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.027

Most of the stability measurement methods [5,9,14,17] calculate stability in terms of code change, however one method [15] calculates stability in terms of code age. The following three main approaches have been used to measure stability: (i) calculate the ratio of the total number of lines added, deleted and modified in a code region to the total number of lines in the code region; (ii) determine the modification frequency of a code region where the modification frequency considers the number of occurrences of consecutive lines added, deleted or modified [9]; and (iii) calculate the average last change dates of cloned and non-cloned code regions using SVN's blame command [15].

1.1. Motivation

The existing stability measurement approaches fail to investigate the following important aspect regarding change.

When comparing the stability of two code regions, it is also important to investigate how many different entities in these two code regions have been affected (*i.e., changed*).

Explanation. We consider two code regions, *Region 1* and *Region 2*, in a software system and each of these two regions has the same number of program entities (suppose 100 entities). If during a particular period of evolution of a software system, the changes that occurred in *Region 1* affected 10 entities while the changes in *Region 2* affected 50 entities, this phenomenon has the following implications.

- **Implication 1 (Regarding change-proneness):** The program entities in *Region 2* are more change-prone than the program entities in *Region 1* for the particular period of evolution regardless of the number of changes that occurred in each of these regions.
- **Implication 2 (Regarding change effort):** The amount of uncertainty in the change process [8] in Region 2 is higher compared to the uncertainty in the change process in Region 1 because, on the basis of the change-proneness of entities during this particular period, while each of the 50 entities in *Region 2* has a probability of getting changed in near future (possibly, in the next commit operation), only 10 entities in *Region 1* have probabilities of getting changed. It is also likely that the requirement specifications corresponding to higher number of entities in *Region 2* are more unstable compared to *Region 1*. Thus, the entities in *Region 2* are likely to require more change effort compared to the entities in *Region 1*.
  This is also possible that the entities in *Region 2* are more coupled than the entities in *Region 1*. In other words, changes in one entity in *Region 2* possibly require corresponding changes to a higher number of other entities compared to *Region 1*. Higher coupling among program entities might cause ripple changes to the entities and thus, can introduce higher change complexity as well as effort.[1]
  Thus, if it is observed that during the evolution of a software system, higher proportion of entities in the cloned region were changed compared to the proportion of entities changed in non-cloned region, then it is likely that cloned region required higher change effort than non-cloned region for that subject system.

Considering the above two implications regarding stability we introduce a new measurement metric: *change dispersion*. We calculate *change dispersion* using method level granularity. The definition of change dispersion with related terminology will be presented in the next section.

1.2. Objective

We perform an in-depth investigation of change dispersion with the central objective of gaining insight into the relative change-proneness (i.e., instability) of cloned and non-cloned code during software maintenance, and investigating ways to minimize the change-proneness of clones. Intuitively, high change-proneness may indicate high maintenance effort and cost. Also, frequent changes to a program entity has the potential to introduce inconsistency in related entities. As such, change-proneness may have important implications for software maintenance. Existing studies regarding clone impact have resulted in controversial outcomes using a variety of different metrics. Our proposed metric, change dispersion, measures an important characteristic of change that has not been investigated before. We perform a fine-grained analysis of clone impact using change dispersion and answer six research questions presented in Table 1.

1.3. Findings

On the basis of our experimental results on 16 subject systems covering four different programming languages (Java, C, C#, and Python) considering three major types of clones (Type-1, Type-2, and Type-3) involving two clone detection tools (CCFinderX[2] and NiCad [24]) we answer the six research questions. The answers (elaborated in Table 9) can be summarized as follows.

[1] Coupling among Entities: http://www.avionyx.com/publications/e-newsletter/issue-3/126-demystifyingsoftware-coupling-in-embedded-systems.html.
[2] CCFinderX: http://www.ccfinder.net/ccfinderxos.html.