Code Clone Tracer (CCT): A Tracking Tool for Analyzing Human and Social Factors in Creating and Reusing Code Clones

Yusuke Kukita
Graduate School of Systems Engineering
Wakayama University
Wakayama, JAPAN
kukita.yusuke@g.wakayama-u.jp

Kojiro Noguchi
Graduate School of Systems Engineering
Wakayama University
Wakayama, JAPAN
noguchi.kojiro@g.wakayama-u.jp

Masao Ohira
Graduate School of Systems Engineering
Wakayama University
Wakayama, JAPAN
masao@wakayama-u.ac.jp

***Abstract*—This paper introduces a tool called CCT (Code Clone Tracer) which is designed to help practitioners and researchers analyze the impacts of human/social factors on software quality in creating and reusing code clones. CCT tracks a history of created and reused code clones for all the revisions in a distributed version control system such as Git. The tool also has features to analyze the relationship between bug-inducing changes and developers, working together with a bug tracking system such as Bugzilla and JIRA. This paper reports our pilot case study where CCT is applied to three open source projects (RxJava, c:geo, and Jackson databind). The results of the case study indicated that a small number of particular developers were involved in many of clone changes to resolve issues.**

***Index Terms*—code clone tracking, software evolution, social network analysis, software quality, open source software development**

I. INTRODUCTION

A number of studies on code clone detection [1]–[3] and analysis [4]–[6] have been reported in the past two decades to address concerns over the code clone management and maintenance. Especially in recent years, some research groups [7]–[9] have been focusing on the propagation process of code clones in long-term evolving systems and its impacts on software quality. In oder to analyze the clone propagation process, [7], [9] have proposed methods for efficiently tracking code clones which were created and reused in the past. However the methods assumed to be used to analyze the clone propagation process recorded in a centralized version control system (CVCS) such as Subversion. It is required to extend them to analyze the code clone propagation process managed in a widely-used distributed version control system (DVCS) such as Git and it is the first motivation for us to develop a new tracking tool. We are also strongly interested in uncovering

- *who has been involved in creating and reusing code clones?*,
- *how did they efficiently or inefficiently manage and maintain propagating clones in an evolving system?*,
- *wha kind of impact does the involved developers and management styles have on software quality?* and so forth

because only a limited number of studies reported social/human factors in creating and reusing code clones.

In this paper, we propose a new code clone tracking tool called CCT (Code Clone Tracer). CCT can track the code clone propagation process recorded in a DVCS reporsitory. In addition, CCT has features to analyze the relationship between bug-inducing changes and developers, working together with a bug tracking system such as Bugzilla and JIRA. In this paper, we also report our pilot case study where relationships between developers and code clones in three open source projects (RxJava, c:geo, and Jackson databind) are visualized and analyzed using a dataset created by CCT.

II. CCT: CODE CLONE TRACER

This section describes a tool called CCT (Code Clone Tracer) which is aimed at helping practitioners and researchers analyze human/social factors in the process of creating and reusing code clones. Since CCT is intended to be used for tracking code clones from a software change history consisting of up to tens thousands of revisions in a distributed version control system such as Git, it is implemented using a coarse-trained clone detection technique [7], [9] to reduce the time required to detect and track code clones from the entire revisions. The tool also has features to analyze the relationship between bug-inducing changes and developers, working together with a bug tracking system (e.g., JIRA). In what follows, we introduce each feature of CCT in detail.

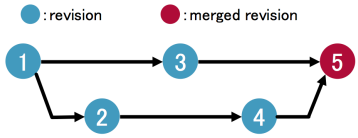
A. Code Clone Tracking

CCT has the following process to provide a feature for tracking a history of code clone creations and reuses recorded in a distributed version control system (DVCS)[1].

1) Identifying parent-child relationships among revisions in Git: Traditional clone tracking tools [7], [9] trace back revisions in a centralized version control system (CVCS) in chronological order and identify code clones in each revision. Since CCT targets at tracing revisions in a DVCS where

[1]Currently, CCT only supports Git repositories. It is still under construction for other kinds of distributed version control systems

978-1-7281-1651-8/19/$31.00 ©2019 IEEE
SNPD 2019, July 8-11, 2019, Toyama, Japan



Parent revision	Child revision
1	2
1	3
2	4
3	5
4	5

Fig. 1: Parent-child relationships of revisions in a distributed version control system

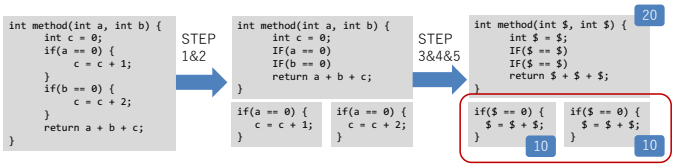


Fig. 2: Detected, normalized, and hashed blocks

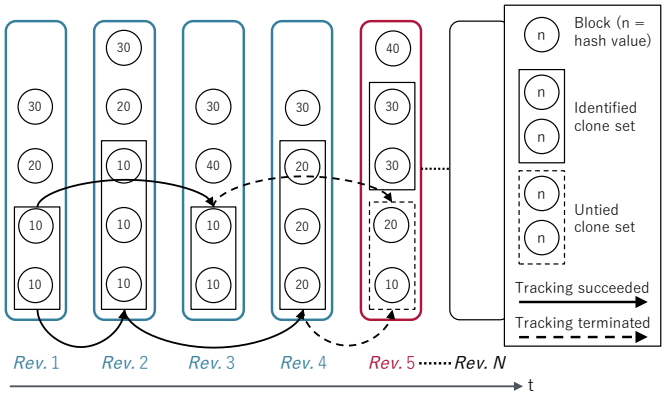


Fig. 3: Code clone tracking in CCT (the revision number corresponds to Figure 1.)

revisions are often be branched and merged intricately, it needs to identify parent-child relationships of revisions as illustrated in Figure 1. First, CCT acquires revision (commit) IDs in a target repository. Second, it identifies parent-child relationships for each revision as in the left of Figure 1. Finally, it records the parent-child relationships as a table (the right of Figure 1). The parent-child table is used to refer to identify a revision with multiple parents and children in the later processing.

2) Identifying all the code fragments in each revision: In this process, source files to extract code fragments from each revision are identified based on the information of each commit. First, CCT acquires the information about revision IDs, committed date and time, committers' IDs (names) and changed (added/modified/deleted) source files from commit logs. Second, all the source files included in the first revision are identified for extracting all the code fragments (i.e., blocks in this paper) in the later step. Third, beginning at the first revision, changed source files are identified by sequentially tracing back revisions to the latest revision. A revision created by a marge commit is often detected during identifying changed source files, because CCT analyzes a DVCS repository. If a merge commit has no conflict, through comparing two commits right before the merge commit, changed source files are regarded as target files for extracting code fragments. If a merge commit has a conflict, all the source files included in the revisions created by the merge commit are regarded as the target files since the process until resolving the conflict is not recorded occasionally. Finally, as illustrated in Figure 2 (STEP1 & STEP 2), code fragments are extracted as blocks with syntactic and lexical analysis for identified target files in each revision.

3) Identifying code clones: All the code fragments in all the revisions are identified as blocks in the previous process. The identified blocks are normalized (STEP 3 in Figure 2) by replacing variable names or literals as special characters such as $ so that code clones only having different variables names or literals can be detected. Then, hash values are calculated based on strings in normalized blocks (STEP 4 in Figure 2). Finally, blocks with the same hash value are regarded as a set of code clones (STEP 5 in Figure 2). The minimum length of tokens is set to 30 in this study. All the code clones in each revision are identified by applying the above processing to all the code fragments.

4) Tracking code clones: In the code clone tracking process, in the first place, hash values of all the blocks in the first revision are compared each other and blocks with the same hash values are packed as a clone set. As illustrated in Figure 3, hash values of clone sets are compared in a sequential order. If a hash value of a clone set in one revision is equal to that after the revision, the two clone sets are linked each other. Linked clone sets between different revisions define the child-parent relationships. At this moment, numbers of elements (clones) in a clone set are also checked to know if they are increased, decreased, or same. If a clone set with a equal hash value cannot be found in a child revision, tracking the clone set is terminated. For instance in Figure 3, tracking the clone set with 10 (hash value) is terminated at *Rev.*. However, only comparing hash values before/after one revision can easily fail to track. Even if hash values do not match before/after one revision, clone sets sometimes can be tracked with the similarity of CRD (Clone Region Descripter) [7]. For instance, the hash value (10) of the clone set in *Rev.*2 changed to 20 in *Rev.*4, but it can be linked using the similarity of CRD (Similarity=0.9 in this study). As with [9], CCT also uses the similarity of CRD to track clone sets between different revisions.

B. Associating Code Clones with Developers and Issue reports

In addition to the feature for tracking code clones in a DVCS repository, CCT associates tracked code clones with involved developers and related issues.

1) Associating clones with involved developers: Developers who involved in creating and reusing code clones are (1) identified, (2) classified into *creators* who created original code clones, *users* who reuse existing clones at different places, *modifier* who modified clones without hash value changes,

TABLE I: Dataset obtained by using CCT

(∗) The total number of developers includes developers who have never changed code clone but have changed other parts of source code.

Project		RxJava	c:geo	Jackson databind
Analysis period		2014/08/30 ∼2016/06/29	2011/07/11 ∼2016/07/21	2011/12/23 ∼2016/07/21
Num. of revisions		1,397	9,821	3,260
Num. of developers	Total (∗)	73	107	114
	Developers who changed clones to fix issues	18	28	17
	Developers who just changed clones	17	27	11
Num. of clone sets	Total	289	2,660	1,610
	Clone sets related to issues	104	2,008	1,223
	Clone sets not related to issues	185	652	387
Num. of reported issues	Total	1,892	5,849	1,307
	Issues with clone changes	35	405	241
	Issues without clone changes	1,857	5,444	1,066

and *eliminators* who deleted clones, and (3) associated with revisions by using committers' ID in commit logs.

2) Identifying bug-induced revisions: CCT has a feature to apply the SZZ algorithm [10] to a issue tracking system such as Bugzilla and Jira. Working together with a issue tracking system, CCT can identify developers who induced bugs, i.e., developers are associated with issue reports and related revisions.

3) Associating clones with issue reports: Based on the result of applying SSZ to an issue tracking system, it can identify if file changes to resolve an issue result in clone changes. By linking clone changes to reported issues, code clones, developers and reported issues are associated each other in order to support a deep analysis on the process in creating and reusing code clones.

III. A Case Study on Developers' Interactions

This section describes a case study where CCT is applied to analyze the relationship between developers' interactions and software quality in a clone change history.

A. Goal and Motivation

It is well-known that interactions among developers have a close relation to software quality. [11] found that there is latent close sub-communities in a large-scale successful OSS project and particular developers belonging to the sub-communities are involved in changes to the same source files. [12] revealed that source files co-authored by unspecific multiple developers likely have more defects than when source files are authored by a single developer. The two studies above were not contradict to each other, but they indicated that active interactions among developers have an impact on software quality. While [11], [12] investigated interactions among developers in file level changes, [8] focused on clone level changes. [8] analyzed the differences of inconsistent changes and related bugs between single-authored and multi-authored clones and showed that there are no clear differences between them.

The case study aims at continuing this line of work to deeply understand the impact of interactions among developers on code clone changes and software quality. Although [8] only counted the number of developers involving in co-authored clone changes, we are more interested in looking at the interaction side in clone changes and we also would like to demonstrate CCT is useful for the case study.

B. Dataset

In the case study, interactions among developers involving in code clone changes are analyzed. We collected data from three active open source projects (RxJava, c:geo, Jackson databind) on GitHub[2]. Table I shows our dataset obtained by using CCT. Please note that "issue" in Table I does not only mean a particular type of issue such as "bug" but it also includes "improvement", "enhancement" and so forth since a project on GitHub can freely tag an issue to be managed and/or resolved in the project. In the case study we do not distinguish issue types, but in the future we need to manually read and classify issues for further analyses.

From Table I, we can confirm that many of changes to code clones (i.e., added/modified/deleted code clones) are made to resolve issues. For instance, c:geo has 2,660 clone sets in total and they have changed 2,008 clone sets to resolve issues. On the other hand, we can also confirm that not so many issues are related to clone changes. Jackson databind has issues associated with clone changes at a higher rate than other two projects, but it is only 18.3% (241/1,307).

C. Analysis

In the case study, interactions among developers in changing code clones are visualized and analyzed using a social network analysis tool so called Pajek[3] as the following procedure.

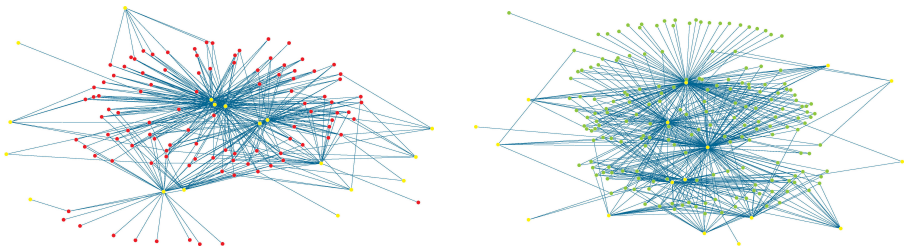
1) Separating issue related clone sets from other clone sets: First, clone sets associated with issues and clone sets not associated with issues are separately identified by CCT.

2) Identifying developers who involved in clone changes: Next, developers who were involved in clone changes (add/modify/delete) are identified by CCT. At the same time, for the comparison, developers who were involved in not-clone changes are also identified.

3) Creating networking data: Using the above information obtained by using CCT, network data in Pajek data format is created through associating clone sets with developers involved in clone changes.

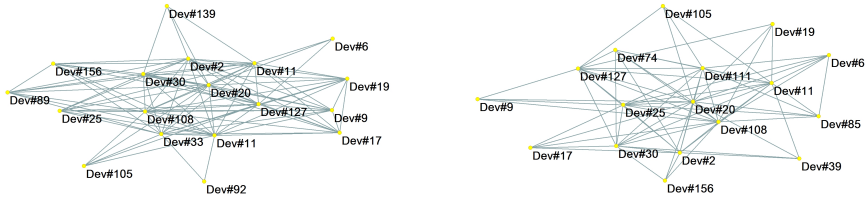
[2]https://www.github.com/

[3]Pajek, http://vlado.fmf.uni-lj.si/pub%20/networks/pajek/



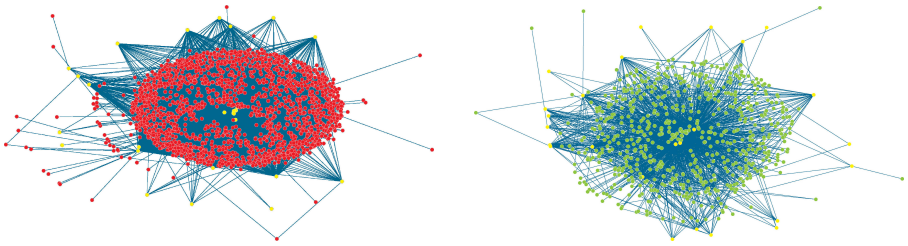
(a) Issue related (Density: 0.603) **(b)** Not Issue related (Density: 0.394)

Fig. 4: Code clones and developers (RxJava)



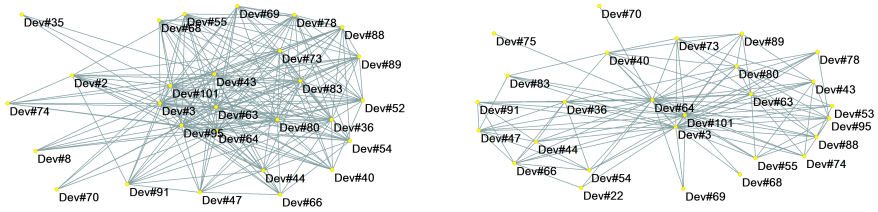
(a) Issue related (Density: 0.614) **(b)** Not Issue related (Density: 0.529)

Fig. 5: Interactions among developers (RxJava)



(a) Issue related (Density: 0.891) **(b)** Not Issue related (Density: 0.333)

Fig. 6: Code clones and developers (c:geo)



(a) Issue related (Density: 0.597) **(b)** Not Issue related (Density: 0.351)

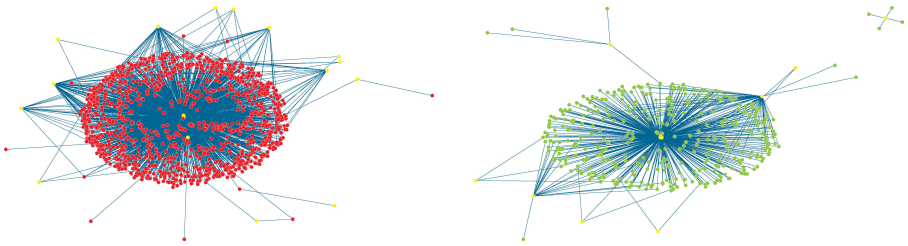
Fig. 7: Interactions among developers (c:geo)

4) Visualizing relationships between clone sets and developers: The network data created in the previous step is visualized as an undirected graph to understand relationships between clone sets and developers.

5) Visualizing interactions among developers: The network data is alternatively visualized as an undirected graph to understand interactions among developers who changed the same clone sets.

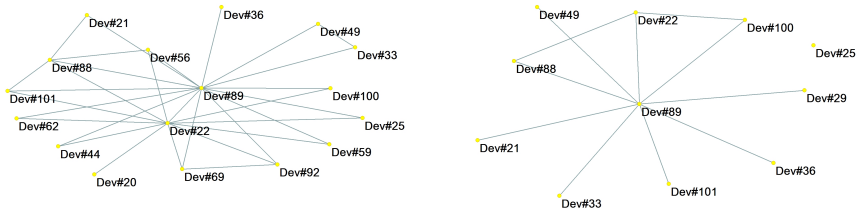
D. Results

As a result of running CCT on a Windows server with 6 core CPU (2.5GHz) and 64 GB memory, the time to detect code clones and to track them were respectively within an hour for a Git repository of each project. Although we only checked tracking results of RxJava manually, the failure rate of code clones tracking was 10.18%. Based on the previous study [9] of a code clone tracking method, this failure rate is considered not so bad.



(a) Issue related (Density: 0.855) **(b)** Not Issue related (Density: 1.223)

Fig. 8: Code clones and developers (Jackson databind)



(a) Issue related (Density: 0.235) **(b)** Not Issue related (Density: 0.231)

Fig. 9: Interactions among developers (Jackson databind)

Figure 4 $\sim$ 9 show results of visualization using Pajek. In all the visualizations, red, green and yellow nodes mean clone sets related to issues, clone sets not related to issues, and developers respectively. Figure 4, 6, and 8 visualize the relationships between clone sets and developers involved in clone changes by connecting developers who changed clones to changed clone sets as edges. Two visualizations are prepared to compare a network in which developers changed issue related clone sets with a network in which developers just changed clone sets not associated with issues. Figure 5, 7, and 9 focus to visualize developers' interactions only. They are created by connecting developers who have changed the same clones set as edges. Interactions among developers means they have collaboratively created/changed/reused the same clone sets.

The score of network density is put down with the sub-captions in each figure. The network density is calculated by dividing connected edges by potentially connectable edges in a graph. The higher density in Figure 4, 6, and 8 indicates developers changed many clone sets in the past, while the lower density indicates developers were only involved in particular clone changes. The higher density in Figure 5, 7, and 9 indicates developers changed the same clone sets, while the lower density indicates developers tend to change separate clone sets. The next section discusses the visualization results and usefulness of CCT.

IV. Discussions

A. Visualization result (c:geo)

In what follows we only discuss the visualization result for c:geo due to the page limitation, but we also observed similar trends in RxJava and Jackson databind.

From Table I, we can confirm that c:geo has the largest number of revisions and clone sets among the three projects. Although it is difficult to visually understand the network structure from Figure 6 (a), several, particular developers were involved in most of issue related code clones and therefore the network density in Figure 6 (a) is very high (0.891). Compared with Figure 6 (a), the structure of Figure 6 (b) is not so dense (0.333). Developers seem to separately change clone sets which is not related to issues.

The same thing can be observed from Figure 7 (a) and (b). Although the network density in Figure 7 (a) is much lower (0.597) than Figure 6 (a), it is because particular developers in the center of Figure 6 (a) dedicated their efforts to change most clone sets to collaboratively resolve issues and other developers seem to help them once or twice. On the other hand, the network density in Figure 7 (b) is approximately same (0.351) as Figure 6 (b). Developers seem not to need tight collaborations each other to change clone sets if the changes are not related to resolve issues.

The result is easy to understand intuitively. Code clones are widely spread among various source files in a large-scale software product. Changes to code clones to resolve an issue have an great impact on issue-fixing tasks and require developers to understand the whole structure of the product. In c:geo (and also in RxJava and Jackson databind), a small number of particular developers seem to have a complete picture of the product and play an important role as gatekeeper to sustain software quality in changing code clones.

B. Potential of CCT

In this case study we just visualized and qualitatively analyzed relationships between code clones in systems and involved developers and interactions among developers in

changing code clones, based on the dataset obtained by using CCT. Since CCT is designed to identify and trace code clones for the entire revisions and to associate involved developers and issues with clone changes, it can be also used for time series analysis and quantitative analysis. Figure 4 $\sim$ 9 only show an overview of whole structures in each project, using all dataset obtained by CCT at a time. In the near future, we would like to further conduct case studies to much more precisely capture social/human factors in the clone propagation process through time series analysis and quantitative analysis.

V. Summary and Future Work

In this paper we proposed a code clone tracking tool called CCT (Code Clone Tracer). We also reported our pilot case study on interactions among developers who has been involved in creating and reusing code clones, based on the dataset obtained by applying CCT to there open source projects' repositories. We found that a small number of particular developers dedicated their efforts to resolve most of reported issues resulting in clone changes. Since the case study in this paper is our first trial to deeply understand how code clones should be appropriately managed and maintained for an evolving system, who should most care about clone changes and so forth, we will continue to apply CCT to more repositories and analyze the clone propagation process in detail.

Acknowledgment

We would like to thank Professor Yoshiki Higo and Dr. Keisuke Hotta for useful discussions and advices. CCT reuses the code clone tracking feature of ECTEC [9]. This research is conducted as part of Grant-in-Aid for Japan Society for the Promotion of Scientific Research (17H00731, 18K11243).

References

[1] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. of the Int'l Conf. on Software Maintenance (ICSM '98)*, 1998, pp. 368–377.

[2] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. of the IEEE Int'l Conf. on Software Maintenance (ICSM '99)*, 1999, pp. 109–118.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 7, pp. 654–670, 2002.

[4] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE '11)*, 2011, pp. 311–320. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985836

[5] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proc. of the 31st Int'l Conf. on Software Engineering (ICSE '09)*, 2009, pp. 485–495.

[6] C. Kapser and M. Godfrey, ""cloning considered harmful" considered harmful: Patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[7] E. Duala-Ekoko and M. Robillard, "Clone region descriptors: Representing and tracking duplication in source code," *ACM Trans. on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 1, pp. 3:1–3:31, 2010.

[8] J. Harder, "How multiple developers affect the evolution of code clones," in *Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance (ICSM '13)*, 2013, pp. 30–39.

[9] Y. Higo, K. Hotta, and S. Kusumoto, "Enhancement of crd-based clone tracking," in *Proc. of the 2013 Int'l Workshop on Principles of Software Evolution (IWPSE '13)*, 2013, pp. 28–37.

[10] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR '05)*, New York, NY, USA, 2005, pp. 1–5.

[11] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proc. of the 16th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE '08)*, 2008, pp. 24–35.

[12] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proc. of the Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE '11)*, 2011, pp. 4–14.