A Pull Request Tool based on Sliding Average Method

Yuxuan Wei, Yuan Sui and Bingyan Li
Experience and Devices Department
Microsoft, Beijing, China
Email: {t-yuxwei, yuansui, binyanli}@microsoft.com

***Abstract*—This paper studies a cost-effective method for peer code review. Tool assisted code review is a form of peer code review. However, there is significant amount of human effort involved in peer code review. It is possible to reduce the human effort which can improve the overall software quality. In recent years, Modern Code Review (MCR), a lightweight and tool-based code inspection, has been widely adopted in both proprietary and open-source software systems. Finding appropriate codereviewers in MCR is a necessary step of reviewing a code change. However, little research is known the difficulty of finding codereviewers in a distributed software development and its impact on reviewing time. In this paper, we investigate the impact of reviews with code-reviewer assignment problem has on reviewing time. We find that reviews with code-reviewer assignment problem take 5 days longer to approve a code change. To help developers find appropriate code-reviewers, we propose 𝔻IFF𝔽INDER, a diff-based code-reviewer recommendation approach. We believe that FINDER could be applied to MCR in order to help developers find appropriate code-reviewers and speed up the overall code review process.**

I. Introduction

Software code review has been an engineering best practice for over 35 years. It is an inspection of a code change by an independent third-party or corporated developer to identify and fix defects before integrating a code change into a system. While a traditional code review, a formal code review involving inperson meetings, has shown to improve the overall quality of software product. however, the traditional practice is limited in the adoption to the globally-distributed software development.

Recently, Modern Code Review (MCR), an informal, lightweight and tool-based code review methodology, has emerged as a widely used tool in both industrial software and open-source software. Generally, when a code change, i.e., patch, is submitted for review, the author will invite a set of code-reviewers to review the code change. Then, the code-reviewers will discuss the change and suggest fixes. The code change will be integrated to the main version control system when one or more code-reviewers approve the change. Rigby et al. find that code reviews are expensive because they require code-reviewers to read, understand, and critique a code change. To effectively assess a code change, an author should find appropriate code-reviewers who have a deep understanding of the related source code to well examine code changes and find defects. As a huge amount of code changes must be reviewed before the integration, finding appropriate code-reviewers to every piece of code changes can be time-consuming and labor-intensive for developers.

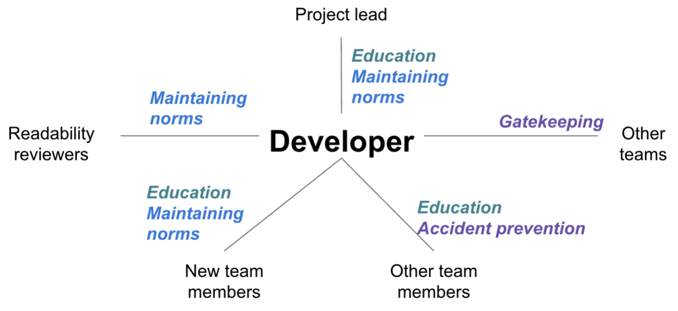


Fig. 1. Reviewer Model

To help developers find appropriate code-reviewers, we propose 𝔻IFF𝔽INDER, a file history-based code-reviewer recommendation approach. We leverage a similarity of previously reviewed history to recommend an appropriate code-reviewer. The intuition is that files that are located in similar file would be managed and reviewed by similar experienced codereviewers.

The main contributions of this work can be summarized as follows,

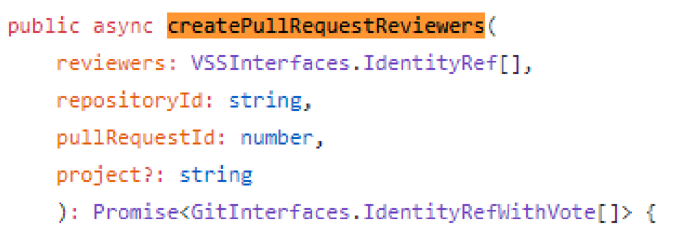
- An exploratory study on the API of Azure DevOps.
- A file diff-based code-reviewers recommendation approach, with promising evaluation results to automatically suggest appropriate code-reviewers for MCR.

II. Background and Example

Code review is the manual assessment of source code by humans, mainly intended to identify defects and quality problems. However, the traditional code review practice is limited in the adoption to the globally-distributed software development. In recent years, Modern Code Review (MCR) has been developed as a tool-based code review system which is less formal than the traditional one. MCR becomes popular and widely used in both proprietary software (e.g., Google, Cisco, Microsoft) and open-source software (e.g., Android, Qt, LibreOffice). Below, we briefly describe the Gerrit-based code review system, which is a prominent tool and widely used in previous studies

To illustrate the Git-based code review system, we use an example of git. In general, there are four steps as following:

1) Author (Smith) creates a change and submits it for review.



```
public async createPullRequestReviewers(
    reviewers: VSSInterfaces.IdentityRef[],
    repositoryId: string,
    pullRequestId: number,
    project?: string
    ): Promise<GitInterfaces.IdentityRefWithVote[]> {
```

Fig. 2. Details of Function

2) The author (Smith) invites a set of reviewers (i.e., Codereviewers and Verifiers) to review the patch.

3) A code-reviewer (Alex) will discuss the change and suggest fixes. A verifier (John) will execute tests to ensure that: (1) patch truly fix the defect or add the feature that the authors claim to, and (2) do not cause regression of system behavior. The author (Smith) needs to revise and re-submit the change to address the suggestion of the reviewers (Alex and John).

4) The change will be integrated to the main repository when it receives a code-review score of +10 (Approved) from a code-reviewer and a verified score of +5 (Verified) from a verifier. Then, the review will be marked as "Merged." Otherwise, the change will be automatically rejected.

From the example, we observe that finding appropriate code-reviewers is a tedious task for developers. The author (Smith) has a code-reviewer assignment problem since he cannot find code-reviewers to review his change. To find code-reviewers, the author (Smith) asks other developers in the discussion:"*Can you please add appropriate reviewers for this change?*" Finding an appropriate code-reviewer can increase the reviewing time and decrease the effectiveness of MCR process. Therefore, an automatic code-reviewer recommendation tool would help developers reduce their time and effort.

III. API Analysis

In this part, we focus on automatic adding reviewers in case we already have a reviewers name list. By analysis the open source API of Azure, we find that there are so many other information we needed when we call the function

$$createPullRequestReviewers. \tag{1}$$

The project ID, repo ID, pull request ID are needed as fig 2. We acquire these informations by analysis the Hook event json file.

Next step we find the reviewers list that needed in (1) is far from the reviewer name list. The unique ID and descriptor for ADD which we can acquire from nowhere but Azure itself. In case of that, we have a prior hypothesis : The reviewers we find must shown in other pull request no matter the roles they play. So we first call the function

$$getPullRequestsByProject. \tag{2}$$

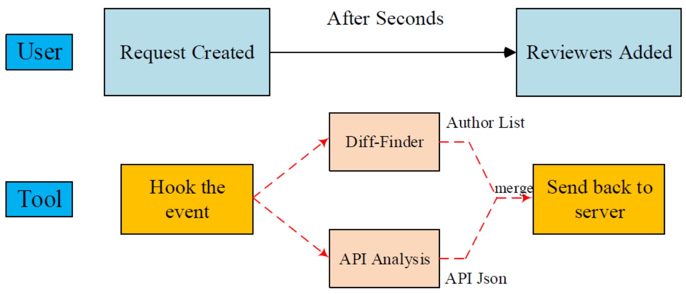


Fig. 3. Structure of a Timeslot

Algorithm 1: Diff Finder

```
Part one: Receive and Calculate;
 Receive target files and base files;
Part two: Diff Finder;
while Diff list is not NULL do
    Find the diff-file element, its index is $m$ ;
    if check $m^{th}$ file's history is TRUE then
        Assign the most nearby history developer
          related to file $m^{th}$ as next reviewer;
        Push back task $m$ to List queue $q_n$;
    else
        continue to next diff file $n^{th}$ ;
    end
end
```

By analysis the response, the reviewers details are found. For effective search, the data structure HASH TABLE is proposed with O(1) time complex.

IV. Diff Finder Algorithm

V. Numerical Results

In this section, we present numerical results to validate the performance of our proposed algorithm as compared to the following banchmark scheme, with a static circumstances. The aim of this evaluation is to investigate the added value of using of the Diff-Finder algorithm.

VI. Conclusion

Acknowledgment

This work was supported by the Min Wan, Dong Shen, Junhao Zheng.