Multi-Perspective Visualization to Assist Code Change Review

Chen Wang, Xiaoyuan Xie*, Peng Liang, Jifeng Xuan
State Key Lab of Software Engineering, School of Computer Science, Wuhan University
Wuhan, Hubei, China
{cnwang, xxie, liangp, jxuan}@whu.edu.cn

***Abstract*—"Change-based" code review plays an important role in open-source project development. Due to the large amount of human involvement and tight time schedule, tools that can facilitate this activity would be of great help. Current tools mainly focus on difference extraction, code style examination, static analysis, comment and discussion, etc. However, there is little support to change impact analysis for code change review. In this paper, we serve this purpose by providing a change review assistance tool, namely, MultiViewer, for the most popular OSS GitHub. We define metrics to characterize code changes from multiple perspectives. Specifically, these metrics mine coupling relations among related files in the changes, as well as estimate the change effort, risk and impact. Such information is visualized by MultiViewer in two formats. We demonstrate the helpfulness of MultiViewer by showing its ability as indicators to some important project features with real-life case studies.**

***Index Terms*—Code review, change impact analysis, visualization, GitHub**

I. Introduction

In the trend of agile software development, "change-based" code review that focuses on small pieces of changes, becomes a very practical approach in software quality assurance [1]. However, due to the great amount of human involvement, how to perform "change-based" code review (referred to as "change review" in this paper) in timely manner is considered as a big challenge [2]. Some commonly adopted ideas focus on implementing tools that extract the code differences, analyze code coverage, check code style, facilitate comments input, and etc. Some well-known tools include CodeReviewHub[1], Gerrit[2], Codecov[3], CRITICS [3], and etc.

As a matter of fact, according to empirical studies from real-life projects, apart from the above facilities, there can be much more information to be presented by a change review tool. For example, comprehension on change coupling relation, warns on the risk of changes relating to defects, estimation on the potential impact from the changes, and etc., are shown to be of much interest to developers from industry [2], [4].

However, current tools provide little support to vividly present such information. Therefore, in this paper, we propose a change review tool, namely, MultiViewer, to serve the purposes. It is built for one of the most popular OSS, GitHub. We formally define three metrics for a commit, namely, *Effort* (costs of making the changes), *Risk* (closeness to bug fixing) and *Impact* (correlation with other components and influence on the entire system). MultiViewer visualizes these information in two forms, one is *Spider Chart* that provides an overview on the commit with respect to the above three metrics. The other one is *Coupling Chart*. Taking the commit under review as the centroid, *Coupling Chart* depicts coupling relations: (1) among all files within the commit; and (2) between files in and outside the commit. We evaluate the helpfulness of MultiViewer with 10 GitHub projects. Through comprehensive analysis, we show that the information revealed by MultiViewer can be good indicators to some project features.

*Corresponding author.

[1]https://www.codereviewhub.com/

[2]https://code.google.com/p/gerrit/

[3]https://github.com/marketplace/codecov

II. Metrics: Effort, Risk and Impact

Before introducing MultiViewer, we first define some metrics to quantify "*Effort*", "*Risk*" and "*Impact*" of a commit. Our metrics extend the basic definitions in [5] and take newly added files into account.

First, we consider the most straightforward metric, "*Effort*". In this paper, we view a commit C as a set of program files $\{f_1, f_2, \ldots, f_m\}$ that have been changed in C.

Definition 1 (*Effort* of commit C, $Effort(C)$)**.** Let $L^C(f_i)$ denote the lines of code that have been changed in f_i by C. Then, we have $Effort(C) = \sum_{f_i \in C} L^C(f_i)$.

$Effort(C)$ aims to reveal the amount of editing work in C, which gives code reviewers the first impression about the changes: the larger the $Effort$ is, the more attentions she/he may be expected to pay.

Suppose a file f has been involved in a sequential list of commits $\overrightarrow{\mathcal{C}(f)}$=<$C_1, C_2, \ldots, C_k$>, and the time stamp of each commit in $\overrightarrow{\mathcal{C}(f)}$ is recorded in the file evolution history as $\overrightarrow{\mathcal{T}(f)}$=<$T_1, T_2, \ldots, T_k$>, where T_i is larger (i.e. later) than T_j if $i < j$. Let $\mathbb{C}(f)$ denote $\{C_1, C_2, \ldots, C_k\}$. Then, the number of times that a file has been changed $FC(f)$ is equal to $|\mathbb{C}(f)|$. A high change frequency of a file may due to various reasons. For example, it may reveal the importance of this file to some extent: a core module of a software may be frequently updated because many other modules depend on it. It may also reveal the low quality of this file: on one hand there might be too many bugs to be fixed; on the other hand frequently changing a file is more likely to introduce future defects.

Intuitively speaking, newly committed changes may have higher influence on the entire project than those remote ones [5]. For example, changes in remote commits may have been overridden by later commits, and files in remote commits even may be no longer available in current version. Thus, we define the influence strengthen of a commit as follows.

Definition 2 (Influence strengthen of commit C_i, $IS(C_i)$)**.** Let T_0 denote the time stamp of current commit C_0 under observation. $<C_1, C_2, \ldots, C_k>$ are a series of commits prior to C_0 with $<T_1, T_2, \ldots, T_k>$. Then, we define the influence strengthen with respect to C_0 of commit C_i as $IS(C_i)=e^{(T_i-T_0)}$, where $T_i<T_0$. The earlier the commit is, the lower its IS is.

As a reminder, to present the attenuation of influence from remote commits, there could be numerous formats apart from the above exponential growth to define $IS(C_i)$. According to our preliminary analysis, the above definition can sharpen the distinction and better support the visualization.

Next, let us consider the risk of commits. To explicitly represent how close a file relates to bug fixing activities, we identify all "bug fixing commits" $\overrightarrow{\mathcal{C}^B(f)}$ in which the file has been involved[4], from $\overrightarrow{\mathcal{C}(f)}$. Let T_0 denote the time stamp of current commit C_0 under observation. For any file $f_i \in C_0$ created before T_0, let $\mathbb{C}^B(f_i)=\{C_1, C_2, \ldots, C_k\}$ represent the set of "bug fixing commits" that f_i was previously involved in. Let $L^{C_j}(f_i)$ denote the lines of code that have been changed in f_i by C_j, and D_i denote the total number of developers who have modified f_i. Then, we have the following definitions.

Definition 3 (*risk* of file f_i, $risk(f_i)$)**.** The *risk* value of a file is defined as $risk(f_i) = D_i * \sum_{C_j \in \mathbb{C}^B(f_i)} (IS(C_j) * L^{C_j}(f_i))$.

$L^{C_j}(f_i)$ records previous efforts made on this file for bug fixing activities, and the influence strengthen is used as a weighting factor. The number of involved developers indicates the extent of concerns on those bugs. Based on this definition, the *risk* value does not directly indicate the suspiciousness of being faulty. Instead, it measures the intensity of the relationships between the file and previous bug fixing activities. The fact that a file has been frequently involved in bug fixing activities implies either this file is of low quality, or it is critical to improve the quality of the entire system.

Definition 4 (*Risk* of commit C_0, $Risk(C_0)$)**.** The risk of commit C_0 under observation at T_0 is the sum of $risk(f_i)$ of all files in C_0: $Risk(C_0) = \sum_{f_i \in C_0} risk(f_i)$.

According to the above analysis, if a commit is with many files of high $risk(f_i)$, it would be necessary to remind code reviewers to pay more attention to the commit.

Finally, we want to estimate the impact of changes, where it is essential to know how the involved files are coupling with the entire system. Let T_0 denote the time stamp of current commit C_0 under observation. If a file $f_i \in C_0$ was created before T_0, we calculate its evolutionary coupling with other files and the entire system as follows.

[4] $\overrightarrow{\mathcal{C}^B(f)}$ are decided via keywords matching.

Definition 5 (Evolutionary coupling between existing files f_i and f_j, $EC(f_i, f_j)$)**.** Suppose f_i and f_j ($i \neq j$) were co-changed in $\overrightarrow{\mathcal{C}}=<C_1, C_2, \ldots, C_k>$ at $\overrightarrow{\mathcal{T}}=<T_1, T_2, \ldots, T_k>$. And the set of these commits is denoted as $\mathbb{C}$. Then, $EC(f_i, f_j)$ with respect to T_0 is defined as $EC(f_i, f_j) = \sum_{C_i \in \mathbb{C}} IS(C_i)$.

This definition is following the intuition that more frequent and recent co-changes lead to closer coupling relation between f_i and f_j.

Definition 6 (Evolutionary coupling between existing file f_i and the entire system, $EC(f_i)$)**.** $EC(f_i)$ with respect to T_0 is $EC(f_i) = \sum_{f_j} EC(f_i, f_j)$, where f_j is any file in the system which has been co-changed with f_i for at least once.

It should be noted that apart from existing files, a commit may also create new files, which have no historical co-change information, thus $EC(f_i)$ is not applicable. For these cases, we calculate the dependent coupling between the new file and the entire system as follows.

Definition 7 (Dependent coupling between new file f_i and the system, $DC(f_i)$)**.** $DC(f_i)$ with respect to T_0 is defined as $DC(f_i) = \sum_{f_j} EC(f_j)$, where f_j is any file in the system, which has dependent relation with f_i. In this paper, we identify dependence between files by checking "import declarations" in a source file.

With the above two definitions, we measure the impact of a commit as follows.

Definition 8 (*Impact* of commit C_0 at T_0, $Impact(C_0)$)**.** $Impact(C_0)$ is the sum of $EC(f_i^{old})$ and $DC(f_i^{new})$ of all files in C_0: $Impact(C_0) = \sum_{f_i^{old} \in C_0} EC(f_i^{old}) + \sum_{f_i^{new} \in C_0} DC(f_i^{new})$, where f_i^{old} is the existing file before C_0, and f_i^{new} is the new file created by C_0.

III. MultiViewer: Visualization of Commits

We implement **MultiViewer** for one of the most popular OSS, GitHub, to visualize the above metrics via *Spider Chart* and *Coupling Chart*.

1) The *Spider Chart* has three spokes, which represent "*Effort*", "*Risk*" and "*Impact*" for a given commit. The data length of each spoke is proportional to the original value relative to the maximum value across all samples of the corresponding metric. Figure 1 shows a sample *Spider Chart* for commit #371e86 for project "commons-lang", whose "*Risk*" value is reaching the maximum; while the "*Impact*" and "*Effort*"values are very close to the maximum. This *Spider Chart* provides a preliminary overview of the commit. For a more detailed inspection, we present the *Coupling Chart*.

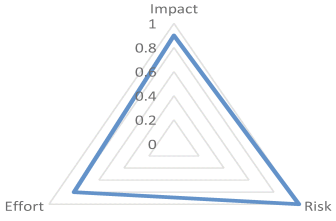

Fig. 1. Sample *Spider Chart* of Commit #371e86 in Repo "commons-lang"

2) The *Coupling Chart* takes the commit under review (C_0) as the centroid, and depicts change coupling relations: (1) among all files within the commit; and (2) between files in and outside the commit. Figure 2 shows a sample *Coupling Chart* for commit #371e86 of project "commons-lang".

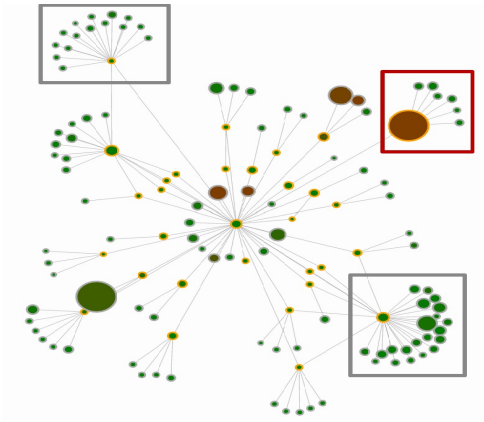

Fig. 2. Sample *Coupling Chart* of Commit #371e86 in Repo "commons-lang"

Each node in the chart represents a file that is directly involved in or related to C_0. There are three features for a node.

- **Aureole.** Each node has aureole in orange or gray, where orange color means that the node (file) belongs to the current commit C_0 and gray color indicates that the node (file) is not directly involved in C_0 but has evolutionary coupling with at least one file in C_0.
- **Size.** Nodes in the chart have different sizes, where a larger circle means more frequent changes on the file. Specifically, we have size_of_node=$FC(f)$+1[5].
- **Color.** The color of a node represents the risk of the file, i.e. $risk(f)$. MultiViewer defines a linear scale that maps the domain of risk values $[0, Max]$ to the range of color ["green", "red"].

Each edge that links two nodes f_i and f_j, reveals the coupling strengthen between them: the length of the edge equals to $1/EC(f_i, f_j)$. In other words, any two linked nodes indicate their previous co-changing. The closer they are from each other, the more frequent co-changes they have encountered. As a reminder, according to our experience, files that are indirectly coupled with each other have their correlations decreased gradually. Therefore, MultiViewer only explores for files directly coupled with the changed files.

To summarize, *Coupling Chart* focuses on a changeset, visualizing its correlation with other relevant files. By navigating these relevant files, developers can better comprehend the changes and estimate their ripple effect on the entire system. For example, Figure 2 shows that there are 31 files involved in this commit. This number of changed files is fairly high among all commits of this project. This explains the high *Effort* (close to the maximum) in Figure 1. And there is one big red node framed by a red rectangular, which indicates that this file has been involved in many bug fixing commits previously. Actually, the risk value of this file contributes a lot to the *Risk* value of the entire commit in Figure 1. This reminds code reviewers to pay more attention on it. It can also be found that files in this commit are coupling with many other files of the system (e.g. node clusters in gray frames), which leads to high *Impact* values in Figure 1.

[5]We add 1 to $FC(f)$ because $FC(f)$ for a newly created file is 0.

IV. Evaluation

We are interested to evaluate the helpfulness of MultiViewer. But as a preliminary analysis, we decided to put comprehensive end-user studies in our future work. In this paper, we investigate whether MultiViewer can reveal any interesting patterns for a commit, such that users can quickly identify some important features by being presented with a figure. In particular, we focus on "group of developer", "type of commit" and "popularity of project", because these features are generally important in review task assignment. We want to address the following research questions:

- **RQ1:** How changes committed by different groups of developers (authors) distinguish from each other with respect to *Effort*, *Risk* and *Impact*. As an open-source community, most projects on GitHub accept contributions from both *Internal* and *External* authors. So, here we investigate how typical *Spider Charts* from different groups of authors look like and whether the project popularity affects the observations.
- **RQ2:** How different types of commits distinguish from each other with respect to *Effort*, *Risk* and *Impact*. In projects of GitHub, commits are made for various purposes. So, here we investigate how typical *Spider Charts* of each type look like and whether the project popularity affects the observations.

A. Data Preparation

1) **Project selection.** We first picked up all Java projects with over 3K commits and at least 200 stars and forks on GitHub. By ranking these projects according to their starred numbers, we selected the top five and the bottom five projects, such that they show high diversity in the popularity. Information for these projects is shown in Table I[6].

2) **Raw data cleaning.** Before the analysis, we processed data cleaning. We first combined commits with the same author and log message within a short period[7]. This is because that there usually exist a sequence of commits with respect to the same task (e.g. to address one issue in a pull-request, a developer may submit multiple commits where the later ones supplement the earlier ones). Secondly, we excluded all "Merge" commits, which contain no changing information but only serve as the heading nodes in three-way merging actions. This filtering process can also help to avoid confusion due to the difference between "committer" and "author".

3) **Author group.** We divided authors into two groups, namely "*Internal*" and "*External*" authors. An author is labeled

[6]Data were collected on May 20, 2017.

[7]In this experiment, we set the period as 24 hours.

TABLE I
SELECTED PROJECTS

Project	Commit	Dev	Star	Fork	Function
BIMServer	3637	15	268	203	Building Information Modelserver
Buck	11123	249	4696	722	A fast build system
Commons-lang	4988	84	960	596	Java libraries for manipulation of Java core classes
Druid	5113	77	6405	3092	A database connection pool
Graphhopper	2990	49	1198	518	A route planning library and server
Hadoop	15993	98	3302	3144	A framework dealing with large data sets
Jackson-databind	3995	109	1274	552	General data-binding package for Jackson
Nutch	2284	24	1188	850	A search engine
Realm-Java	7012	66	7736	1215	A mobile database
Spring-Framework	14739	218	14152	10431	A coupling optimization framework

as "*Internal*" if she/he belongs to the organization that owns the project; otherwise, she/he is an "*External*" author.

4) **Commit type.** We considered seven major types of commits, namely "Cleanup" for deleting code (CLN), "Improvement" for improving performance (IMP), "JavaDoc" for inputing documents (DOC), "Configuration" for configuring the project (CONFIG), "Defect" for fixing bugs (DEF), "Feature" for implementing features (FEATURE) and "Test" for testing code (TEST). We searched for keywords in the commit message to identify its type. For example, keywords for type DEF are "$bug|fix|error|fail|leak|correct|IssueNumber$".

5) **Project popularity.** We divided the 10 projects into two groups, namely *Popular* and *Niche* projects, according to their *Starred* and *Forked* numbers. Specifically, projects with "Star" higher than 3000 and "Fork" higher than 600 are considered as *Popular*; otherwise, *Niche*[8]. Accordingly, we have Buck, Druid, Hadoop, Realm-Java and Spring-Framework as *Popular* projects; and have the rest five as *Niche* projects.

B. About the Author Group (RQ1)

We divided all commits into four groups: G_1 is from *Popular* projects and *External* authors; G_2 is from *Popular* projects and *Internal* authors; G_3 is from *Niche* projects and *External* authors; and G_4 is from *Niche* projects and *Internal* authors. In order to address RQ1, we first identified the representative commits of each group. Within each G_i, we ran clustering algorithm *KMeans* on all commits[9]. We found that in each group, there is a cluster that gathers over 50% of all commits of the group. Specifically, the percentage in G_1 is 71.3%, in G_2 is 70%, in G_3 is 51% and in G_4 is 59.2%. We picked up the centroid of the largest cluster as the representative commit of the corresponding group, whose *Spider Charts* are shown in Figure 3.

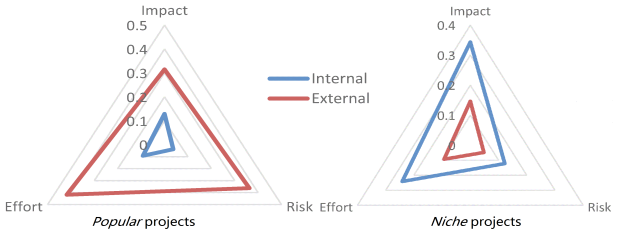

Fig. 3. Typical *Spider Charts* for Different Groups of Authors

To further explain these observations, we adopted "density map" to visualize the distribution of "*Effort*", "*Risk*" and "*Impact*" in different groups, as shown in Figure 4. The left sub-figure compares "*Effort*", "*Risk*" and "*Impact*" between G_1 and G_2; while the right sub-figure presents the comparison between G_3 and G_4. We divided the logarithmic values of all commits into 10 sub-ranges, where *Range* i represents values in $[i-1, i)$ where $1 \leq i \leq 9$; *Range 9+* represents values in $[9, \infty)$. The darkness in each block indicates the ratio of commits within the corresponding sub-range among all commits of the same category: the darker the color is, the higher ratio it is.

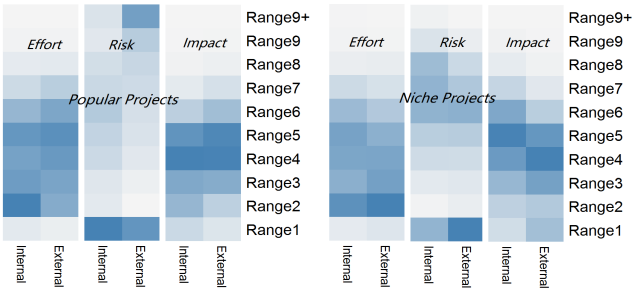

Fig. 4. Comparison Between "*Internal*" and "*External*" Authors

From Figure 4, we can find that in *Niche* projects, *External* authors tend to make small commits (i.e. low *Effort*) than *Internal* authors. In contrast, *Popular* projects present opposite situation. And for *Risk*, in *Popular* projects, commits from *Internal* authors are more likely to have very low "*Risk*" values than those from *External* authors. And *External* authors also contribute high percentage of commits with very large *Risk*. But different comparison results can be found in *Niche* projects. Finally, comparison on *Impact* presents similar results as the above. We also conducted a *Wilcoxon-Mann-Whitney* test with significant level of 0.05, which give consistent conclusion with the observations.

To summarize, we have **addressed RQ1**: In both *Popular* and *Niche* projects, typical *Spider Charts* for commits from *Internal* and *External* authors look quite different. And the relation between commits from *Internal* and *External* authors in *Popular* projects is **opposite to** that in *Niche* projects. In

[8]We set these thresholds according to some general experiences and the actual distributions on these values of the 10 projects.

[9]k is set as 6 based on our preliminary experiments, to make each cluster cohesive.

Popular projects, *Internal* authors make commits with lower *Effort*, *Risk* and *Impact* values than *External* authors. And in *Niche* projects, the conclusion is opposite.

C. About the Commit Type (RQ2)

Similar to **RQ1**, we first identified typical commits for each of the seven types, in both *Popular* and *Niche* projects, and their typical *Spider Charts* are shown in Figure 5.

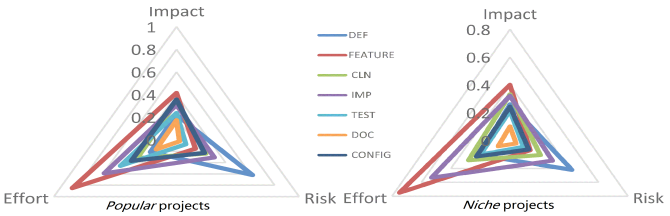

Fig. 5. Typical *Spider Charts* for Different Types of Commits

To further explain these observations, we presented distributions of the three metrics for different commit types, in *Popular* and *Niche* projects, as shown in Figures 6 to 8.

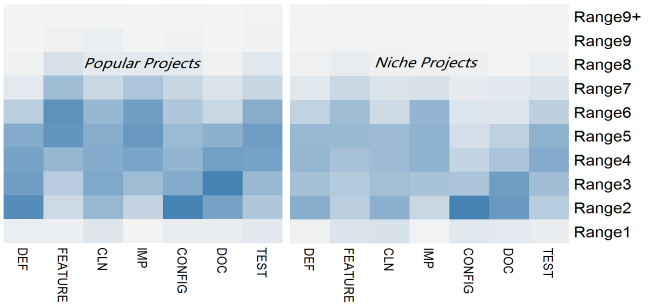

Fig. 6. Comparison on *Effort*

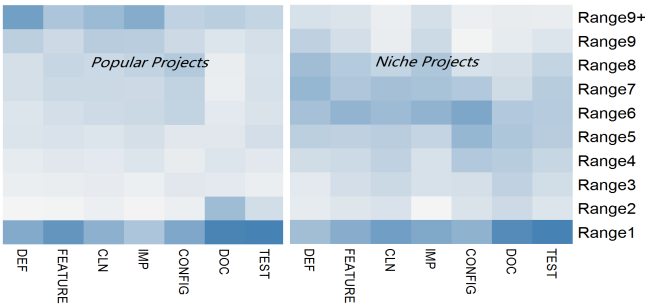

Fig. 7. Comparison on *Risk*

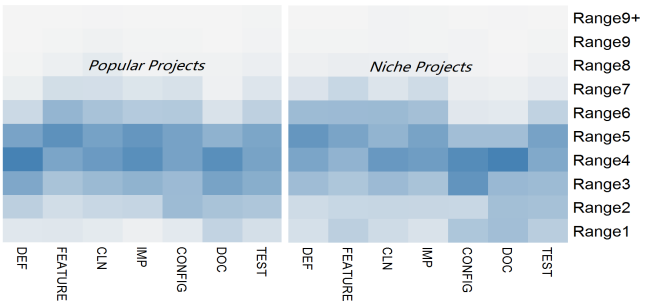

Fig. 8. Comparison on *Impact*

Figure 6 shows that in both *Popular* and *Niche* projects, *Effort* appears quite differently among the seven types of commits. For example, in *Popular* project, "Feature" and "Improvement" commits are more likely to involve high *Effort*; while "Defect", "Config" and "JavaDoc" commits generally require very low *Effort*. *Niche* projects show similar comparison results.

For *Risk*, in Figure 7, the top-2 darkest blocks at "Range 1" are "JavaDoc" and "Test", which indicate that these two types generally involve files not close to previous bug fixing. This is mainly because that these commits usually require more new file creation than existing file modification. In both *Popular* and *Niche* projects, "Defect" commits show polarization on their *Risk* values, which discloses two major types of defect-fixing commits: (1) Low *Risk* values are from commits where the corresponding bug fixing activities mainly rely on adding new files (i.e. patches). These newly added files obviously have no historical "bug fixing" records. According to the definition, *Risk* of these commits are very low. (2) High *Risk* values are from commits where the corresponding bug fixing activities mainly involve modifying files appeared frequently in previous "bug fixing" commits, thus these commits have high *Risk* values. The second case shows that files that have been frequently changed for bug fixing are very likely to be involved in new bugs. This observation suggests that our *Risk* metric can reflect potential risk of being faulty or introducing faults to some extent.

Finally, in Figure 8, *Impact* values of commits in all the seven types from both *Popular* and *Niche* projects present centralization at middle-level of ranges. This is somewhat anti-intuition: we have expected that "Defect" and "Feature" commits show obviously high densities at high ranges; while other types have more commits with lower *Impact*. The reasons will be investigated in our future works.

To summarize, we have **addressed RQ2**: Different types of commits show quite different distributions of their *Effort*, *Risk* and *Impact*, but comparison among different commit types does not show significant difference between *Popular* and *Niche* projects. "Feature" and "Improvement" have high *Effort* in both *Popular* and *Niche* projects; and "Defect" is distinguishable with its high *Risk*, in both types of projects.

D. Threats to Validity

- The primary threat to **internal validity** is about the correctness of MultiViewer and our experiment platform. In order to assure the quality of our implementations, we have conducted thorough testing to improve their reliability and trustworthiness.
- The primary threat to **external validity** is the representative of our results acquired from 10 projects on GitHub. Though the number of projects is not very large, we selected them with high diversity, in order to distinguish them and provide relatively good representativeness. Moreover, we have carefully investigated the rationales behind our observations. Due to the space limitation, we leave this analysis in our follow-up studies.

- The primary threat to **construct validity** is about the metrics for characterizing a commit. Actually, these metrics were defined under commonly accepted intuitions and some in-depth inspections on the data verified their plausibility. In our future studies, we will further refine the metrics for advanced facilities.

V. Related Works

Code review is an important method for software quality assurance. One research direction is to recommend reviewers whose background and skills well match the task [6]–[10]. Other code review assistance includes extracting the code differences, decomposing a large set of tangled changes into small pieces for easier review [11], [12], prioritizing changes for better resource allocation [13], etc. However, as indicated by real-life case studies, disclosing file correlations and presenting change impact are required by code reviewers, which receive little support from current tools [2], [4], [14], [15].

In fact, ***Change Impact Analysis*** (CIA), as a classic research area in software engineering has been studied for many years. Based on various coupling metrics [5], [16], the major task of CIA is to identify a group of co-changed files. Zimmermann et al. proposed ROSE by mining frequent patterns to achieve this goal [17]. Similar studies can be found in [18], [19]. There were also studies for visualizing coupling relations for a project [20]–[22].

As a reminder, the above CIA studies have different purposes with ours, and the visualization techniques are targeted at the entire system. In contrast, MultiViewer focuses on a very small portion of the entire source code, and provides "Just-In-Time" coupling and impact visualization with concise and essential information for code change review.

VI. Conclusions and Future Study

In this paper, we defined metrics for code changes, which reveal coupling relations among related files in the changes, as well as estimate the change effort, risk and impact. We also provided a change review assistance tool for GitHub, namely, MultiViewer to visualize such information in *Spider Chart* and *Coupling Chart*. We demonstrated the helpfulness of MultiViewer by showing its ability as indicators to some important project features.

Our method synthesizes several major techniques in CIA, with tailoring to adapt to our application scenarios. In our future studies, we will further improve MultiViewer from various aspects, such as change completeness checking, defect prediction, and etc. More importantly, a comprehensive end-user evaluation on how well MultiViewer works in real-life development will be conducted. Feedbacks from human studies will be collected to improve the tool.

Acknowledgments

This paper is partially supported by the National Natural Science Foundation of China (Grant No. 61572375, 61472286 and 61502345).

References

[1] T. Baum, O. Liskin, K. Niklas, and K. Schneider, "A Faceted Classification Scheme for Change-Based Industrial Code Review Processes," in *Proceedings of the 2nd International Conference on Software Quality, Reliability and Security*, 2016, pp. 74–85.

[2] L. MacLeod, M. Greiler, M. A. Storey, C. Bird, and J. Czerwonka, "Code Reviewing in the Trenches: Understanding Challenges and Best Practices," *IEEE Software*, vol. PP, no. 99, pp. 1–1, 2017.

[3] T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive Code Review for Systematic Changes," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 111–122.

[4] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry," in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 51:1–51:11.

[5] M. D. Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *Proceedings of the 16th Working Conference on Reverse Engineering*, 2009, pp. 135–144.

[6] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 931–940.

[7] Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Reviewer Recommender of Pull-Requests in GitHub," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, 2014, pp. 609–612.

[8] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. i. Matsumoto, "Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 141–150.

[9] A. Ouni, R. G. Kula, and K. Inoue, "Search-Based Peer Reviewers Recommendation in Modern Code Review," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, 2016, pp. 367–377.

[10] M. B. Zanjani, H. Kagdi, and C. Bird, "Automatically Recommending Peer Reviewers in Modern Code Review," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 530–543, 2016.

[11] K. Herzig and A. Zeller, "Untangling changes," 2011, available: https://www.st.cs.uni-saarland.de/publications/files/herzig-tmp-2011.pdf.

[12] Y. Tao and S. Kim, "Partitioning Composite Code Changes to Facilitate Code Review," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 180–190.

[13] E. v. d. Veen, G. Gousios, and A. Zaidman, "Automatically Prioritizing Pull Requests," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 357–361.

[14] P. C. Rigby and C. Bird, "Convergent Contemporary Software Peer Review Practices," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 202–212.

[15] A. Bacchelli and C. Bird, "Expectations, Outcomes, and Challenges of Modern Code Review," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 712–721.

[16] R. Robbes, M. Lanza, and M. Lungu, "Logical coupling based on fine-grained change information," in *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008, pp. 42–46.

[17] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Proceedings of the 13th International Workshop on Principles of Software Evolution*, 2003, pp. 73–73.

[18] F. Jaafar, Y. G. Gueheneuc, S. Hamel, and G. Antoniol, "An exploratory study of macro co-changes," in *Proceedings of the 18th Working Conference on Reverse Engineering*, 2011, pp. 325–334.

[19] M. Mondal, C. K. Roy, and K. A. Schneider, "Improving the detection accuracy of evolutionary coupling by measuring change correspondence," in *Proceedings of IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, 2014, pp. 358–362.

[20] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *Proceedings of the 13th International Workshop on Program Comprehension*, 2005, pp. 259–268.

[21] D. Beyer, "Co-change visualization," in *Proceedings of the 21st International Conference on Software Maintenance*, 2005, pp. 89–92.

[22] M. D. Ambros, M. Lanza, and M. Lungu, "Visualizing co-change information with the evolution radar," *IEEE Transaction on Software Engineering*, vol. 35, no. 5, pp. 720–735, 2009.