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 CodeReviewHub1, Gerrit2, Codecov3, 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.

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

\textit{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 \textit{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 \(C(f) = \langle C_1, C_2, \ldots, C_k \rangle\), and the time stamp of each commit in \(C(f)\) is recorded in the file evolution history as \(T(f) = \langle T_1, T_2, \ldots, T_k \rangle\), where \(T_i\) is larger (i.e. later) than \(T_j\) if \(i < j\). Let \(C(f)\) denote \{C_1, C_2, \ldots, C_k\}. Then, the number of times that a file has been changed \(FC(f)\) is equal to |\(C(f)\)|. 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.

1https://www.codereviewhub.com/
2https://code.google.com/p/gerrit/
3https://github.com/marketplace/codecov
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” $C^{BI}(f)$ in which the file has been involved4; from $C(f)$. 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 $C^{BI}(f_i)=\{C_0, C_1, C_2, \ldots, C_k\}$ represent the set of “bug fixing commits” that $f_i$ was previously involved in. Let $L^{CI}(f_i)$ 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 $risk(f_i) = D_i \times \sum_{C_j \in C^{BI}(f_i)} IS(C_j) \times L^{CI}(f_i)$.

$L^{CI}(f_i)$ 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.

$\sum_{C_j \in C^{BI}(f_i)} IS(C_j)$ 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 $C := \{C_1, C_2, \ldots, C_k\}$ at $T := \{T_1, T_2, \ldots, T_k\}$. And the set of these commits is denoted as $C$. Then, $EC(f_i, f_j)$ with respect to $T_0$ is defined as $EC(f_i, f_j) = \sum_{C_i \in C} IS(C_i)$.

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 defined as $EC(f_i) = \sum_{j} EC(f_i, f_j)$, 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 $DC(f_i) = \sum_{f_j \in \text{system}} EC(f_i, f_j)$, 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”.

![Coupling Chart](image)

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 = \( FC(f) + 1 \).

- **Color.** The color of a node represents the risk of the file, i.e. \( \text{risk}(f) \). MultiViewer defines a linear scale that maps the domain of risk values \([0, \text{Max}]\) to the range of color \( [\text{“green”}, \text{“red”}] \).

Each edge that links two nodes \( f_i \) and \( f_j \), reveals the coupling strength between them: the length of the edge equals to \( 1/\text{EC}(f_i, f_j) \). 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.

### 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.  

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

\(^5\)We add 1 to \( FC(f) \) because \( FC(f) \) for a newly created file is 0.

\(^6\)Data were collected on May 20, 2017.

\(^7\)In this experiment, we set the period as 24 hours.
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 inputting 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. 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. 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.

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.

### Table I

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

We set these thresholds according to some general experiences and the actual distributions on these values of the 10 projects. $k$ is set as 6 based on our preliminary experiments, to make each cluster cohesive.

---

567
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.

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