Categorizational Study on Commit Purpose For A Better Software Quality

Jincheng He
Department of Computer Science
University of Southern California
jinchenh@usc.edu

***Abstract*—Nowadays, it is common to see software being developed with support from online version control systems (VCS), such as Git. This makes it easier to track the development as well as to evaluate the quality of the software. At the same time, open source makes the projects more vulnerable [] because the developers can be from different organizations with different level of skills. Thus, open source software (OSS) quality investigation, control and improvements are one of the critical issues in the area of software engineering, and repository mining is one of the approaches to conduct them. Thus far, by repository mining, various researches have been conducted to assess the quality of software, but there is still space for investigations about analyzing how developer behaviors impact software quality and to extend the concept of software quality. In this research area, little work has been done in classification-based study on commits, focusing on purposes of commits, commit message and code patterns, to analyze their relations with and impact on software quality. In this white paper, we list fields where and how further research can be done for the sake of a better software quality.**

***Index Terms*—Software Engineering, Software Maintenance, Software Quality, Open Source Software**

I. Introduction

A. Open Source Software and Version Control System

It has been a long time since developing software in an open source repository became a common way of developing software. In those project, some of them are of industrial scale. As the scale has grown far beyond the level that an individual can control and manage, how to efficiently conduct quality control and project management is one of the critical issue in open source software development.

Most industry-scale software are developed by iterative contributions from the project teams [], through ICSM, Agile, DevOps or other process models. In the iterations, version control systems, such as Git and SVN, play a critical role by enabling and facilitate the concurrent contributions from developers. Each revision, or commitment (referred as "commit" in the rest of the paper) contains diffs which are the lines developers change.

These changes can be made by developers from different area of the world, at different times, with different purposes and have different level of impacts on the software [1], correspondingly having negative or positive impact on the software quality. Thus, it is necessary to investigate how these differences influence the software quality to understand it and control it better during the development and maintenance phases.

Focusing on the different purposes of commits, this research white paper investigate how different types of commits, with respect to their purposes, impact the software quality and propose what we can do to control it.

B. Level of Commit Impacts

In all commits to a project, some commits can be less while others being more impactful on the project and software quality. Some projects have multiple modules, one of which is core modules while others are less critical to the entire project. Moreover, some commits many contains only a few documentation fixes while others do hundreds of lines of modifications. These commits in this situation can have different level of impact on the entire project.

The level of impact can be defined in various ways to specify what to investigate. For example, in some previous study, researchers defined impactful commits by whether they are in the core module [2], [3]. We believe that the more critical are the commits, the earlier they should be taken care of, in the sense of quality control and management.

C. Purposes of Commits

While the level of impact can be different, the type of commits can also be different. For example, some commits add a few lines of documentation or comments to code while others can refactoring the code structure or made module-level modifications.

It is common for developers to upload single-purpose commits[1]. However, in commits where changes such as refactoring, adding new dependencies, minor fixing happen, the commits tend to grow beyond its intended task. Each commit includes a prescriptive message documenting the changes made[2], in practice with varying degrees of efficacy.

In addition, it has been shown that different types, by purpose, of commits have impact on one aspect of the software quality, which is compilability [1]. Thus, it is worth working on to investigate how different types of commits impact other aspects of software quality and how they are related to the other metadata of software to acquire a new series guidelines by which developers can following to help improve the overall software quality.

[1]https://www.freshconsulting.com/atomic-commits/
[2]https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project

Previous research has produced and refined taxonomies for commit categorization [1], [4]–[9] which is an option for this research for understanding the purposes of commits.

D. Commit Message

One critical piece of the project metadata is the commit message. When developers push some change to the online repository, a commit message is usually required to explain what they have change in the commit they want to push.

These messages provide important clues for understanding the purposes, thus the types of those commits. As a result, in order to conduct this categorizational study, a further insight is needed in analyzing commit message, about how they are written by human, how they are generated by some agents and how they are related to the software quality, for example, readability and maintenability. [TODO: Some more references to be added here.]

E. Code Pattern

The commit messages are summaries from developers about their work while the code is more fundamental and significant part of the commits. How code should be written, for the sake of better software quality, has long been a hard problem to attack. [TODO: Add some more references here.] A good example of this kind of work can be Dr. Liscov and her contribution to code hierachy.

Also, there are some work aiming at how to generate code automatically, which can be seen in testing code generation tools [TODO: Some more contents to be added here.]

Both hand-typed code and auto-generated code are related to this research. For hand-written code, it is possible to use them to train a prediction model by using NLP techniques to automatically extract keywords from code changes, thus supporting the categorization. On the other hand, auto-generation can be one of the applicational practice of the research results, since one of the goal of the research is to build guidelines of how to better contribute to software with respect to different types of changes.

F. Software Quality

The final goal of the research is to improve software quality in the sense of how to contribute to open source software. Thus, defining the way to assess software quality of projects is one of the most important issues of this research.

As we know, software quality can be evaluated from different directions. For example, COCOMO and COCOMO II [TODO: Add some more contents and references here] evaluate software with respect to their cost. Some tools, such PMD, SonarQube, FindBugs and CAST evaluate software by defining metrics based on software metadata and some algorithms, reflecting the level of security, vulnerability and architecture. [TODO: Some more references here, including the official site links].

In this research, the tool-based software quality metrics is major representation of quality we plan to use. In a more general sense, we use compilability [TODO: Add something about robustness, maintenability, feasibility], and on the completion of this research, plan to use categories and category distributions of commits as a novel set of software quality metrics.

G. Outline of This Research White Paper

In this white paper, we evaluate previous work on close research areas of this research: commit message, code parttern and software quality representation and propose a purpose-oriented categorizational analysis approach on commits and software quality.

Our goal is to construct a reasonable categorization for commits, based on their purposes, investigate its relation with software quality and consequently find a way to improve the software quality.

The sections of this white paper is organized as following:

- Section II-A introduce our current data set, which we will use for the research, following by a plan of what we want to collect in addition to current set to support further analysis.
- Section III discusses previous works in categorizing commits, either by their purposes, sizes and other criteria and how we can apply them to analyze impact on software quality. Furthermore, it also discusses the critical issues in the process of categorization that need extra effort to deal with.
- Section IV discusses previous works in analyzing, extracting information, and auto-generation of commit messages [TODO: some more sub-directions can be added here] and how we can use commit message to help categorization and analyzing quality metrics.
- Section V discusses previous works in code patterns in the context of natural language processing and how we can use them to support the categorization and how they are related to software quality.
- Section VI discusses our way of assessing software quality, including tool-based analysis and extra new metrics.
- Section VII discusses the final goals of this research and how it can be applied practically to serve software development for a better quality.
- Section VIII concludes the white paper.

II. Data

A. Current Data Set

Samples are filtered from the dataset introduced in our previous studies [2], [3], which provides compilability and software quality metrics across 68 open-source Java projects owned by Apache, Google, and Netflix. This study is performed on 914 commits, 314 uncompilable and 600 compilable.

We use the following terminology, as shown in Fig. 1, to categorize commits based on their impact on compilability. A core module is a module that contains the majority of the source code, such as the main modules in most Apache library systems. A commit is *impactful* if it changes a core module. An impactful commit is *broken* if it creates an uncompilable

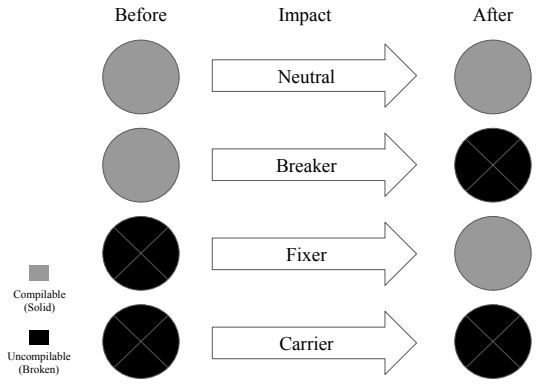

Fig. 1. Four Types of Impactful Commit

revision; otherwise, it is *solid*. A broken commit is a *breaker* if it breaks the compilability of its solid parent; otherwise, it is a *carrier*. A solid commit is a *fixer* if it fixes its broken parent; otherwise it is *neutral*. Fig. 2 shows an example of a commit sequence with these four types of impactful commit.

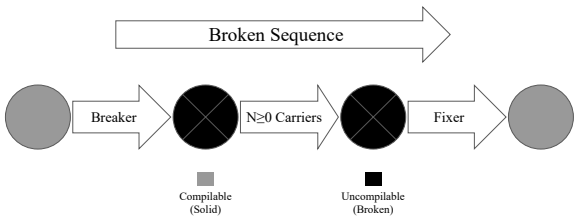

Fig. 2. An Example of a Broken Sequence

We do not categorize orphan or merge commits into breaker, carrier, fixer, or neutral as their impact is not identifiable by comparing two software revisions (i.e., before and after).

For selecting subject systems, we retrieve all Java projects owned by Google and Netflix from GitHub. We select a system only if it 1) requires Ant, Maven, or Gradle for compilation and is not an Android, a Bazel, or an Eclipse project, 2) does not require manual installation of other tools (e.g., Protoc) for compilation, 3) is an official product of the organization, and 4) has a core module containing a substantial amount of code. We target the core module and identify impactfuls in each system and exclude those with fewer than 100 distinct revisions. For selecting Apache subject systems, we use the same criteria as Netflix and Google, except that we only consider subject systems with fewer than 3000 commits by April 2017.

To develop the commit purpose taxonomy, we analyze 100 commits, using both the code and commit message in deriving the categorization. We consult with Hindle et al. to clarify category definitions, which help to determine the categories included in our final taxonomy. In order to apply this taxonomy to small commits, we create further refinements of the definitions for four kinds of tasks.

We use the resulting categorization taxonomy to tag our dataset of 914 commits. Each commit is labelled and cross-validated by multiple individuals. Initial inconsistencies in tagging arise from ambiguities in the original taxonomy. To resolve these ambiguities, we study each inconsistent tag, identify the source of confusion and further refine the taxonomy definitions. The tag definitions are now more narrow in scope, and overlapping meaning between category definitions is reduced. This methodology ensures that our taxonomy can be applied consistently across varied tagging interpretations.

We also define a threshold for identifying commit size, and use this to label each commit – *large* or *small*. Some related works have introduced thresholds to differentiate between large and small commits; however, our dataset exhibits a narrower range in commit size. The average commit size is also reduced due to our focus on the changes within a commit, as opposed to cumulative commit size. Thus, based on our dataset, we create a new threshold which defines commit size as the sum of added and deleted lines of code. The 314 uncompilable commits were split into two parts, with the smallest 157 commits labeled *small*, and the remaining 157 commits labeled *large*. The resulting threshold – 184 lines of code – is used to label the 600 compilable commits. This threshold is based on the distribution of uncompilable commits in order to motivate discussions on the relative sizes and categorization of uncompilable commits. The final dataset contains 157 each of large and small uncompilable commits, and 138 & 462 large and small compilable commits, resp.

To analyze software quality evolution over uncompilable commits in relation to the quality of compilable commits, we examine metrics on code complexity, maintainability, and security, provided by SonarQube [3] and PMD [4]. As uncompilable commits cannot be directly assessed by dynamic software quality analysis, we measure the overall quality change in the compilable commits before and after uncompilable sequences, extrapolating to determine the individual quality of each commit.

B. Additional Data to be Collected

III. Commit Types

A. Previous Works

Previous works primarily characterize commits based on metadata. Alali et al. [5] define commit properties by size – lines of code, number of code blocks, and file count – as well as extracted terms from commit messages. Kaur et al. [10] propose a classifier for labeling commit type – bug repair, feature addition, and general – based on commit messages. However, commit message alone is not enough to effectively categorize commits. While a commit message can indicate developer intent, it will not necessarily address all the major code changes. Dragan et al. [6] perform categorization over commit code, stereotyping added and removed methods to

[3]https://www.sonarqube.org/
[4]https://pmd.github.io/

form a descriptor for commit change. Solely focusing on code to determine commit characteristics can often introduce additional noise. While commit messages are brief summaries indicating the central goal of the commit, code diffs capture details that do not affect high-level categorization. Further, determining commit type, based entirely on code, becomes difficult on extremely large commits, where a commit message may provide all of the information needed.

Hindle et al. [4] propose a taxonomy based on the maintenance tasks to categorize large commits. This categorization uses maintenance attributes first introduced by Swanson et al. [7]. They further map the categories to the taxonomy of Mauczka et al. [8], dividing changes into high-level classes of software maintenance: *adaptive*, *perfective*, and *corrective*. The taxonomy is then used for automatic categorization [9]. Based on the results and methodology, our work proposes a refinement of the taxonomy presented by Hindle et al, adapted to reduce ambiguity between categories, and to support tagging small commits.

B. A Refined Categorization

To analyze the difference in purpose between breakers and neutrals, we need an accurate categorization for commits. Although we leverage Hindle's categorization, it was originally designed for usage only on large commits. Several change types such as Maintenance, Bug Fix, Debug and Refactoring were too broadly defined when applied to smaller commits, and at times needed to be accompanied by other categories to fully classify the commit change.

Our analysis leads to a refinement of Hindle's categories, and the identification of the commits that warrant defining new categories for comprehensive classification. These modifications result in variations in category frequency from those found by Hindle et al. [4]. This indicates, in part, significant changes in category definitions. For example, out of the 2000 commits that they analyzed, 2% were tagged as maintenance while in our analysis it's more than 40%. This result is surprising, as other studies [11]–[13] report that 75% of software development budgets are dedicated to maintenance. Another contributing factor for the variation in category frequency is our inclusion of small commits. Along with the new taxonomy, we provide results that describe the relationship between commit size and purpose.

Table I shows the refined taxonomy. Specifications for types are as follows:

Bug Fix: In small commits, the lack of descriptive code makes it hard to differentiate between Bug Fix and Maintenance categorization. As a result, we have to rely on commit messages and project change logs. For example, if a commit message mentions fixing an existing issue, it should be tagged a Bug Fix. We also thoroughly examine previous software revisions, to see if the commit is added in response to flaws in recent code. While we take these steps to better identify Bug Fixes, some commits are still too vague to be clearly tagged Maintenance or Bug Fix. Further improvement and definition

TABLE I
REFINED CATEGORIZATION

Type	Hindle's Definition	Our Explanation
Branch	If the change is primarily to do with branching or working off the main development trunk of the version control system.	
Bug fix	One or more bug fixes.	A change that is reported in the developing log, change log or with expressions in commit message that indicates it is a correction of unexpected behavior.
Build	If the focus of the change is on the build or configuration system files. (such as Makefiles).	
Clean up	Cleaning up the source code or related files. This includes activities such as removing non-used functions.	
Legal	A change of license, copyright or authorship.	
Cross	A cross cutting concern is addressed (like logging).	
Data	A change to data files required by the software (different from a change to documentation).	
Debug	A commit that adds debugging code.	
Documentation	A change to the system's documentation.	
External	Code that was submitted to the project by developers who are not part of the core team of the project.	
Feature Add	An addition/implementation of a new feature.	New function/methods/class implemented, impacting functionality.
Indentation	Re-indenting or reformatting of the source code.	
Initialization	A module being initialized or imported (usually one of the first commits to the project).	
Internationalization	A change related to its support for languages other-than-English.	
Source Control	A change that is the result of the way the source controls system works, or the features provided to its users (for example, tagging a snapshot).	
Maintenance	A commit that performs activities common during maintenance cycle (different from bug fixes, yet, not quite as radical as new features).	Functional changes without feature add and do not have evidence to be considered a bug fix, including performance Improvement and feature extension.
Merge	Code merged from a branch into the main trunk of the version control system; it might also be the result of different and non-necessarily related changes committed simultaneously to the version control system.	
Module Add	If a module (directory) or files have been added to a project.	
Module Move	When a module or files are moved or renamed.	
Module Remove	Deletion of module or files.	
Platform Specific	A change needed for a specific platform (such as different hardware or operating system).	
Refactoring	Refactoring of portions of the source code.	Relocation of part of code. Restructuring. Extract a part of code out of a function and create a new function.
Rename	One or more files are renamed, but remain in the same module (directory).	
Testing	A change related to the files required for testing or benchmarking.	
Token Replace	An token (such as an identifier) is renamed across many files (e.g. change the name or a function).	
Versioning	A change in version labels of the software (such as replacing "2.0" with "2.1").	

of rules are needed to make the tagging more definitive on small commits.

Feature Add: We found that Hindle's definition of a Feature Add is under-specified for smaller commits. This may be because new features are easily identifiable in larger commits via the commit message, whereas in small code changes, new features can be a single non-utility function or method. We include this in our refined definition of Feature Add.

Maintenance: This is the most under-specified type in the original taxonomy. Cross validation indicates more than half of the commits with inconsistent tags are with Maintenance. Thus, we consult with Hindle et al. regarding this tag. According to our discussion, this should be better translated to "minor perfective changes" which is different from the maintenance tasks we generally use. Since the original definition can cause confusion due to little specification, we define it as function & efficiency improvements, additions of utility functions, or minor modifications without new methods or error corrections. For example, memory cleaning is tagged as Maintenance because it is a performance improvement.

Refactoring: We specify Refactoring as relocation, restructuring of code without altering how the code functions, such as extracting code to form a new function, relocating code hunks within or across files, or other changes to reduce code redundancies. They do not change functionality or efficiency.

Note that by our definition, breakers and neutrals can not be merge or orphan commits, as we study the impact of a commit by comparing two revisions: the one it changes and the one it produces. As a result, there are no commits in the Branch and Initialization categories.

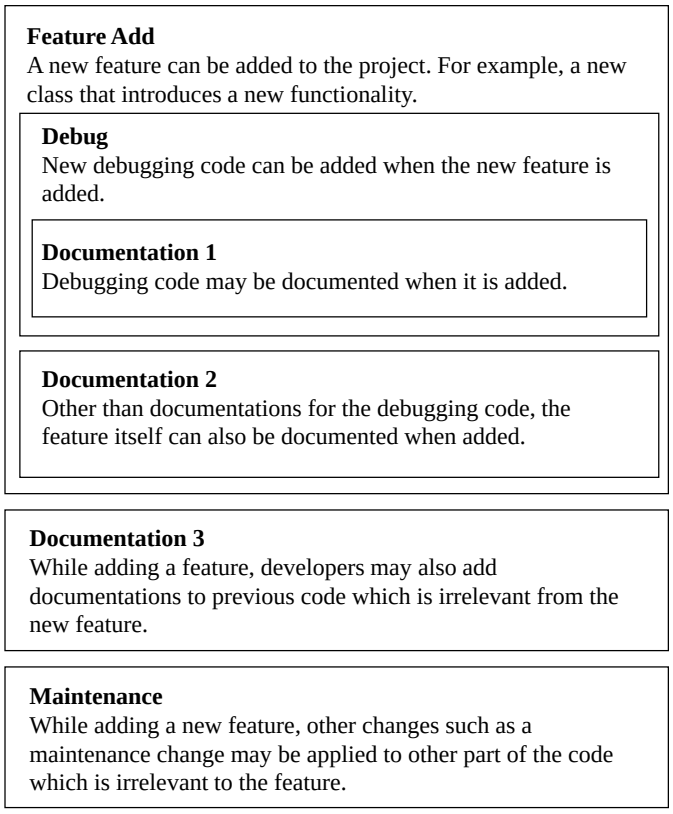

Fig. 3. Explanation For Independent Change

C. The Difference Between Categories with Respect to Software Quality

D. Independent Change

To improve our tagging, we first note that the original change type definitions overlap. For example, a documentation change may occur as part of a feature add commit. This increases tagging ambiguity and complexity, which we solve by introducing the concept of an *independent change*.

We call a change an *independent change* when it is not a part of other changes. Fig. 3 explains this concept. It illustrates a commit with multiple changes, in which a new feature contains three sub-changes. Each block represents a code change. In the figure, a new feature change contains debugging code, documentation 1 for the functionality of the feature, documentation 2 for the debugging code, an irrelevant documentation 3, and an irrelevant maintenance change. In this case, the debugging code and documentation 1 and 2 are sub-changes to the new feature. As a result, we only assign a "Feature Add" tag to the new feature instead of assigning tags to all sub-changes. As documentation 3 and maintenance are both unrelated to the new feature, they will be assigned Documentation and Maintenance tags. The new feature, documentation 3, and maintenance groups are independent changes, with the associated three independent tags forming a label the commit. With the introduction of independent change types, we reduce the ambiguity in the initial taxonomy which results in fewer cross-validation errors and improved tagging efficiency.

E. Single-tagged Commit And Multi-tagged Commit

When reviewing manual tagging, we analyze commits with different numbers of tags separately. The commits are divided into two groups: *single-tagged* and *multi-tagged*. *Single-tagged* commits contain only one independent change, and *multi-tagged* more than one.

Fig. 4 shows the tag-distributions of single and multi-tagged commits by commit purpose. In the figure, black bars represent the rates when a certain commit is tagged a single category while gray bars are used to denote multi-tagged. For example, approximately 80% commits are tagged Testing and more than 95% of them are multi-tagged commits. As shown in the figure, Build, Feature Add, Indentation, Refactoring, Testing arise more frequently in multi-tagged commits, indicating these tags tend to appear together with other tags. More than 95% of testing commits have multiple tags, which is consistent with development practices of including the testing code along with most changes. Documentation is more frequent in single-tagged commits, indicating that many documentation changes are added independently. As the change of documentation is usually used to explain changed code, this may also imply that

contributors often forget to add enough documentation when they accomplish a commit. For other categories there are no apparent differences between single-tagged and multi-tagged commits. In addition to comparing these two sets, we also study how many tags each commit has and its distribution.

1) Tag Distribution: We analyze the differences between the breaker set and the neutral set in terms of tag distributions. Recall that we call a commit a breaker if it breaks the compilabilty of a compilable revision. If a commit changes an uncompilable revision and produce another uncompilable revision we do not consider it as a breaker. We also call a commit a neutral if it changes a compilable revision and produces another compilable revision. If a commit fixes compile errors of an uncompilable commit we do not consider it as a neutral.

Fig. 5 shows the tag-distributions of breakers and neutrals with regard to their commit purpose. Each bar stands for the occurrence rates of that category. For example, around 40% of breakers and neutrals are tagged Testing. Since a commit could have multiple independent changes, values for one color may add up to more than 100%. The presence ratios for Bug Fix, Documentation, is higher in the neutrals while Feature Add, Build, Refactoring, Clean up and Maintenance is higher in breakers. The breakers have 1.90 tags on average while neutrals have 1.56 on average.

TABLE II
RATIO OF COMPILABLE COMMITS OVER NEUTRALS AND BREAKERS

Category	Compilablity Neutral(%)	Breaker(%)	p-value
Branch	0.00	0.00	-
Bug fix	18.17	9.55	**0.000**
Build	3.67	12.10	**0.000**
Clean up	10.00	13.38	0.150
Cross	1.67	1.27	0.781
Data	0.00	0.00	-
Debug	1.17	0.32	0.276
Documentation	13.50	6.05	**0.005**
External	0.00	0.00	-
Feature Add	16.83	27.71	**0.000**
Indentation	3.50	2.55	0.552
Initialization	0.00	0.00	-
Internalization	0.17	0.00	1.000
Legal	0.67	1.91	0.101
Maintenance	41.83	55.10	**0.000**
Merge	0.17	0.00	1.000
Module Add	0.00	0.32	0.344
Module Move	0.17	1.59	**0.020**
Module Remove	0.17	1.27	**0.050**
Platform Specific	0.00	0.00	-
Refactoring	4.33	11.78	**0.000**
Rename	0.00	1.27	**0.014**
Source Control	1.00	1.91	0.358
Testing	36.83	39.17	0.518
Token Replace	2.00	2.87	0.486
Versioning	0.33	0.00	0.549

Again, we perform odd ratio test (the Fisher's Exact Test) to study the correlation between categories and compilability. According to the test result shown in Table II, we found that Bug Fix, Build, Documentation, Feature Add, Maintenance, Module Move, Module Remove, Refactoring, Rename have significant differences between breakers and neutrals, which is consistent with the results of Fig.5.

F. Single-tagged Commit And Multi-tagged Commits

Recall that we call a commit with multiple independent changes *multi-tagged* and a commit with only one independent change *single-tagged*.

Almost 50% of our analyzed commits are multi-tagged. We summarize the number of tags for each commit independently for the breaker and the neutral set, as shown in Fig. 6. The figure shows the distributions of commits with different numbers of tags in the breakers or neutrals. Each bar stands for the ratio of presence of commits with certain number of tags.

For example, 55% of neutrals only have one tag. The curves are Kernel Density Estimations of these two distributions that are long tail distributions. The number of instances decreases as the number of tags increases. As for its relation with compilability, we can see a neutral is more likely to have a small number of tags while a breaker has a relatively large one. We also use the Kernel Density Estimation (KDE) to estimate these two distributions. The distribution density curve of neutrals is sharper than breakers which means the proportion of commits decreases more sharply in the neutrals than in the breakers with increase of the number of tags. We can interpret it as a commit is more likely to break its compilability when it has more tags.

1) Small And Large Commits: We also analyze the relation between uncompilability and whether a commit is large or small. We draw boxplot of changed Lines of Code (LOC) for breaker set and neutral set which is shown in Fig. 7. Fig. 7(a) shows the distributions of neutrals and breakers respectively. (b) shows distributions of commits with different number of tags respectively. In boxplot, each box represents inter-quartile range (IQR) of data points, the range of major part of data and circle represents outlier of data points which are outside 1.5 IQR. As shown in Fig. 7(a), the distribution of breakers is shifted upward from the distribution of neutrals and it means breakers tend to have larger changed LOC than neutrals. This indicates larger commits are more likely to result in compilation breach.

We also analyze the distribution of changed LOC for commits which contains different number of tags as shown in Fig. 7(b). The result shows that if a commit contains more tags, it tends to have more lines of code changed.

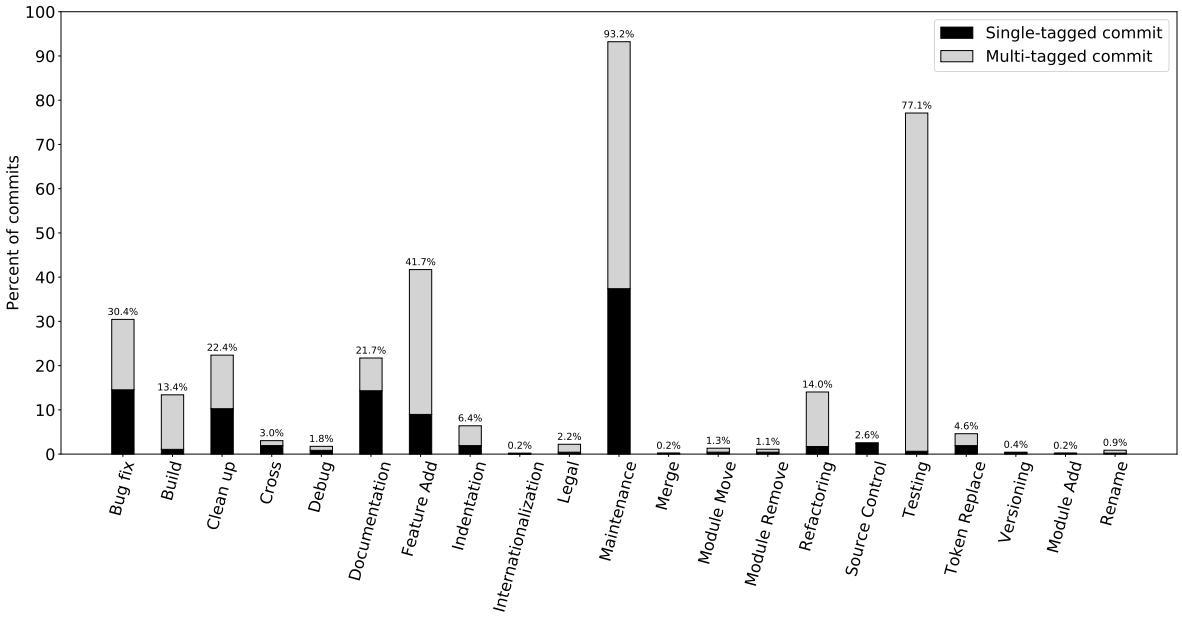

Fig. 4. Change Type Distribution Between Single-tagged and Multi-tagged Commit

IV. Commit Message

A. Initiatives of Commit Messages

B. Preprocessing Removal of Useless Information in Commit Messages

C. The Relation Between Commit Message and Software Quality

V. Code Patterns

A. Different Patterns Existing in Code.

B. Different Patterns May Have Different Impact on Software Quality

VI. Software Quality

A. Tool-based Quality Metrics

To understand how software quality evolves over uncompilability, we identify all the broken sequences in our dataset. Each broken sequence starts with a breaker and ends with a fixer. The broken commits in the sequence cannot be analyzed using software quality tools. However, we can compare software quality between two revisions to understand how software quality evolves over the sequence: the revision produced by the impact-parent of the breaker and the one produced by the fixer. The former is the last solid revision before the sequence begins and the latter is the first solid revision after the sequence ends.

B. Compilability

A software revision created by a commit is expected to be compilable. However, uncompilability can occur due to careless development — failure to compile the software locally prior to pushing to the shared repository. It can also result from variations in build environments, incompatibility across overlapping changes made by multiple developers, or changes in upstream dependencies. The presence of compile errors inhibits bytecode and dynamic software analysis, as well as static analysis when the code is unparsable [2]. Previous studies [2], [14]–[16] have shown that even in popular open-source projects maintained by major software organizations, build-breaking commits can occur.

Behnamghader et al. [3] explore the qualitative properties of uncompilable commits. Further insight into the types of commits that most frequently cause build errors can help to inform better development practices. Additionally, this correlation can be used as a part of future methods for predicting and preventing uncompilable commits. Analyzing the degradation of software quality over multiple uncompilable commits highlights the long-term negative impact of careless development, and the importance of fixing build errors immediately after they take place. Also, understanding the purposes of those uncompilable commits can result in preparing guidelines to avoid such degradation in the future.

Compilability. Multiple recent studies [**?**], [**?**], [14]–[18] have assessed the compilability of software repositories. To our knowledge, none address the effects of developer purpose on uncompilability, or how software quality evolves when the code is uncompilable. Hassan et al. [14] focus on automatically building the last commit for the top 200 Java repositories on GitHub. Seo et al. [17] analyze 26.6 million builds produced over a period of 9 months by Google engineers, reporting the build failure frequency and cause, as well as how long

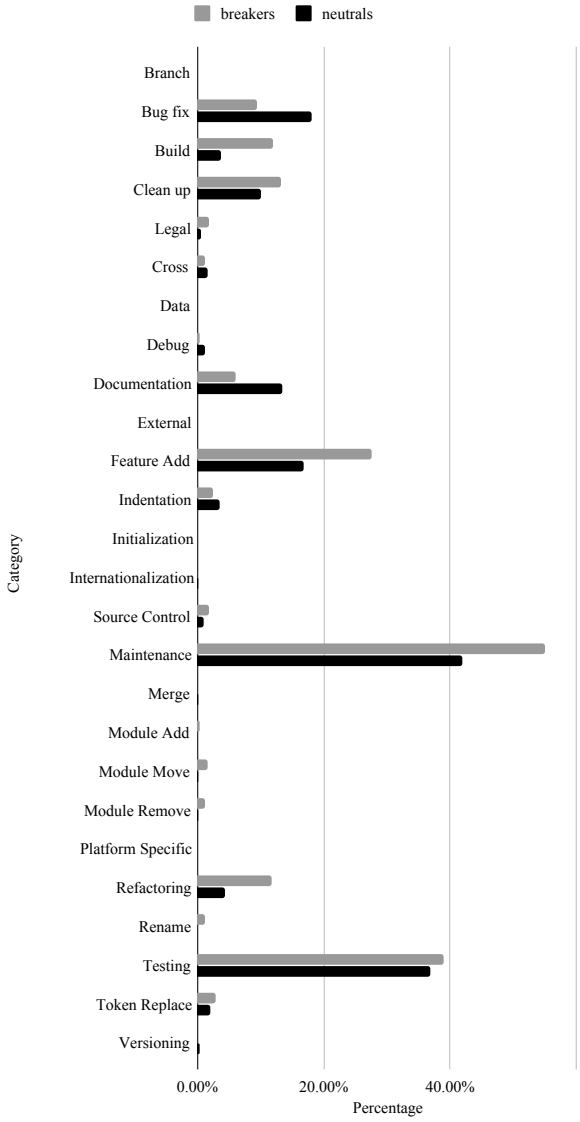

Fig. 5. Tag-distribution for Neutrals and Breakers

it takes to remediate. Hassan et al. [15] propose a build-outcome prediction model, based on combined features of build-instance metadata and code changes, to predict whether a build will be successful. Macho et al. [18] identify 125 commits in 23 repositories that repair a missing dependency, qualitatively and quantitatively analyze how the fix is applied, and propose an approach to fix dependency build breakage automatically. Tufano et al. [16] study the compilability of 219,395 snapshots of 100 Java projects from the Apache Foundation, analyzing the frequency and possible causes of broken snapshots. Benamghader et al. [**?**] qualitatively study why developers commit uncompilable code, and design an approach [**?**] to increase compilation ratio over commit history and to study sequences of uncompilable commits in terms of their length and interval.

C. Other Quality Aspects

VII. Goal

VIII. Conclusions

A. Development Guidelines

B. A Novel Set of Software Quality Evaluation Metrics

This direction mainly focuses on investigating the developer behaviors including how they contribute to the software project, especially to open source software where we can get enough metadata for analysis to evaluate how different behaviors impact the software quality and how to improve this process for the sake of better software quality. Specifically, this research will focus on modelling the change pattern in code when contributors make the commits and how the code impacts the quality metrics. In this direction, research has been conducted to investigate when, where, how and what the developers contribute to the projects but there is still space for research from the coding side. To reveal details, for example, the change types and contents, in code level can reveal further detail how different changes related to developer behaviors and how they impact the software quality, which is represented in the software quality metrics.

This direction could succeed since the current techniques in machine learning, statistics, natural language processing will be sufficient to support this research. Once it succeeds, we can provide further instructions in coding, if possible, more reliable coding standards which can lead to an improvement and standardization of coding in the software engineering area.

This direction may take more than ten years since it could be a gradual improvement. The success of this direction depends on how the applied techniques evolve in the coming years. The midterm milestone could be a reasonable high prediction accuracy while the final exam milestone could be the completion of the new systematic coding standards.

Research has been done to investigate the impact of some behaviors of developers when they contribute to software, especially open source software. However, in the previous research in this area, they haven't reveal the correlation between the change type and the code as well as their impact on the software quality. In this research, we start with categorization of commit changes in open source software repositories and show their impact on the software quality, collected by using different static analysis tools, by which we get software metrics for each revision of the software. In the end of this paper, we propose a new categorization for the commit change, based on reading the code and the commit message. We also apply neural network to train a model to predict the potential software quality changes after a certain kind of change with an accuracy of 90

In previous research, we have been proven there exist correlation between the change type and the software quality, which is represented by the software quality metrics. However, this is not enough to put this research into valuable application. In this research, we refine the categorization by applying a natural language processing approach to the code to parse it.

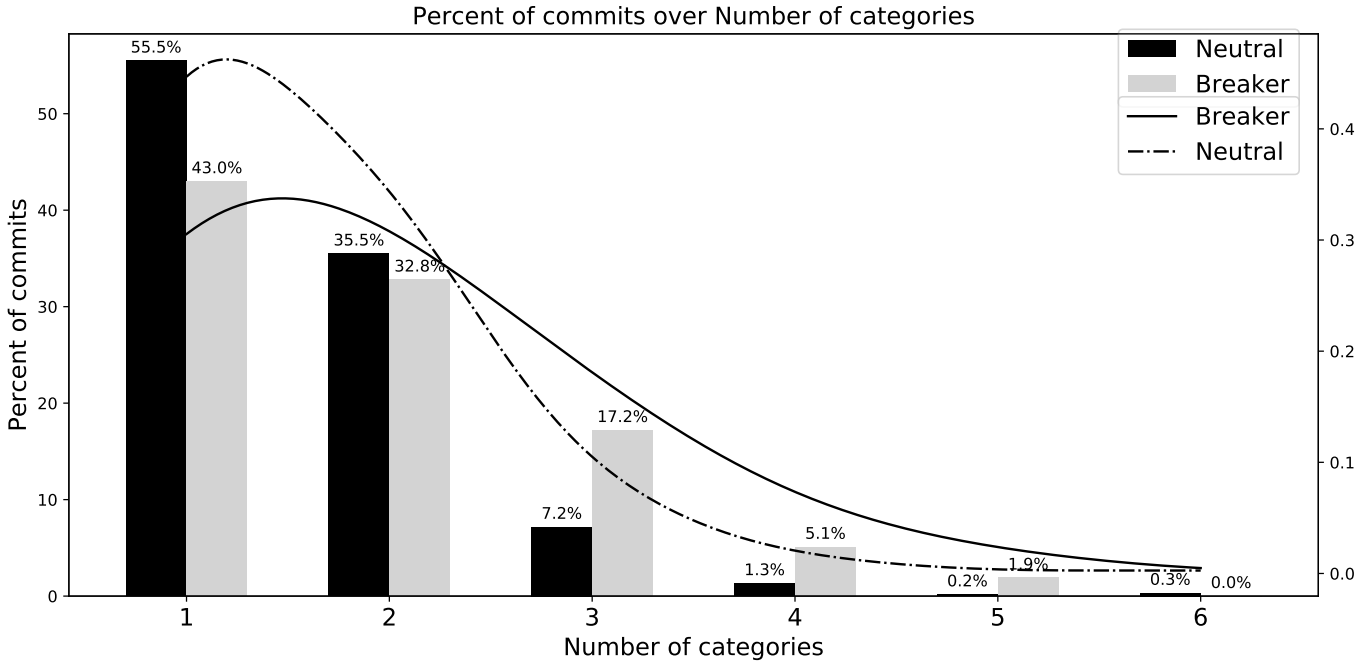

Fig. 6. Compilability With Number of Tags in Each Commit

In this way, we can model the change into a lower level, and make more accurate prediction of what is happening in the repository of the open source software. The results shows we can predict the change of software quality change based on the code with a high accuracy of 95

In previous research, the researchers use natural language processing methods to predict the possible changes to the software quality based on the code changes. This means we can provide guidelines to some extent for software contributors when they commit to a software. However, the previous study mainly focus on training the model with current data which has its limitation. In this research, we collect new data from different software repositories with more than 100 coding styles, including most of existing coding styles. By comparing the different coding styles and training the new prediction model, we come up with a guideline for how to code more efficiently with less software quality issues. The results turn out to be valuable, providing a 95

ACKNOWLEDGMENT

This material is based upon work supported in part by the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract No. HQ0034-13-D-0004 Research Task WRT 1016 – "Reducing Total Ownership Cost (TOC) and Schedule." SERC is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology.

REFERENCES

[1] J. He, S. Min, K. Ogudu, M. Shoga, A. Polak, I. Fostiropoulos, B. Boehm, and P. Behnamghader, "The characteristics and impact of uncompilable code changes on software quality evolution," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, 2020, pp. 418–429.

[2] P. Behnamghader, P. Meemeng, I. Fostiropoulos, D. Huang, K. Srisopha, and B. Boehm, "A scalable and efficient approach for compiling and analyzing commit history," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18. New York, NY, USA: ACM, 2018, pp. 27:1–27:10. [Online]. Available: http://doi.acm.org/10.1145/3239235.3239237

[3] P. Behnamghader, R. Alfayez, K. Srisopha, and B. Boehm, "Towards better understanding of software quality evolution through commit-impact analysis," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2017, pp. 251–262.

[4] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: A taxonomical study of large commits," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370773

[5] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *2008 16th IEEE International Conference on Program Comprehension*, June 2008, pp. 182–191.

[6] N. Dragan, M. L. Collard, M. Hammad, and J. I. Maletic, "Using stereotypes to help characterize commits," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 520–523.

[7] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2Nd International Conference on Software Engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 492–497. [Online]. Available: http://dl.acm.org/citation.cfm?id=800253.807723

[8] A. Mauczka, M. Huber, C. Schanes, W. Schramm, M. Bernhart, and T. Grechenig, "Tracing your maintenance work – a cross-project validation of an automated classification dictionary for commit messages," in *Fundamental Approaches to Software Engineering*, J. de Lara and A. Zisman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 301–315.

[9] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *2009 IEEE 17th International Conference on Program Comprehension*, May 2009,

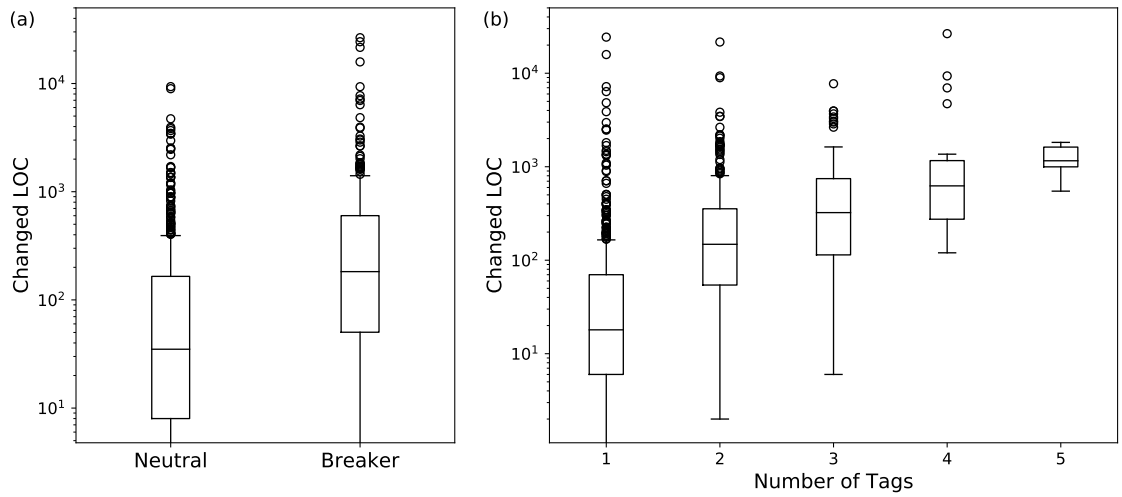

Fig. 7. Commits Distribution Over Lines of Code.

pp. 30–39.

[10] A. Kaur and D. Chopra, *GCC-Git Change Classifier for Extraction and Classification of Changes in Software Systems*. Singapore: Springer Singapore, 2018, pp. 259–267.

[11] G. L. Dodaro, "Government efficiency and effectiveness: Opportunities to reduce fragmentation, overlap, and duplication and achieve other financial benefits," GOVERNMENT ACCOUNTABILITY OFFICE WASHINGTON DC, Tech. Rep., 2015.

[12] Q. Redman, "Weapon system design using life cycle costs," *Raytheon Presenatation*, 2008.

[13] J. Koskinen, "Software maintenance fundamentals," *Encyclopedia of Software Engineering, P. Laplante, Ed., Taylor & Francis Group*, 2009.

[14] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang, "Automatic building of java projects in software repositories: A study on feasibility and challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov 2017, pp. 38–47.

[15] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov 2017, pp. 157–162.

[16] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017, e1838 smr.1838. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1838

[17] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 724–734. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568255

[18] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 106–117.