How to Explain a Patch: An Empirical Study of Patch Explanations in Open Source Projects

Jingjing Liang*†, Yaozong Hou‡, Shurui Zhou§, Junjie Chen¶, Yingfei Xiong*† and Gang Huang*†
*Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, PR China
†Department of Computer Science and Technology, EECS, Peking University, Beijing, PR China
‡Beijing Kuaishou Technology Co., Ltd, Beijing, PR China
§Carnegie Mellon University, Pittsburgh, USA
¶College of Intelligence and Computing, Tianjin University, Tianjin, PR China
*{jingjingliang,xiongyf,hg}@pku.edu.cn, ‡hyz200926@yeah.net,
§shuruiz@andrew.cmu.edu, ¶junjiechen9208@gmail.com

***Abstract*—Bugs are inevitable in software development and maintenance processes. Recently a lot of research efforts have been devoted to automatic program repair, aiming to reduce the efforts of debugging. However, since it is difficult to ensure that the generated patches meet all quality requirements such as correctness, developers still need to review the patch. In addition, current techniques produce only patches without explanation, making it difficult for the developers to understand the patch. Therefore, we believe a more desirable approach should generate not only the patch but also an explanation of the patch.**

To generate a patch explanation, it is important to first understand how patches were explained. In this paper, we explored how developers explain their patches by manually analyzing 300 merged bug-fixing pull requests from six projects on GitHub. Our contribution is twofold. First, we build a patch explanation model, which summarizes the elements in a patch explanation, and corresponding expressive forms. Second, we conducted a quantitative analysis to understand the distributions of elements, and the correlation between elements and their expressive forms.

***Index Terms*—patch, explanation, program repair, bug-fixing**

I. Introduction

Bugs are inevitable in software development and maintenance, and bug fixing constitutes one of the largest portions of maintenance cost [29]. To reduce the cost of bug fixing, a lot of automated approaches have been proposed to repair bugs [16], [32]. These approaches usually generate a patch and apply it to the buggy program to make the program satisfy certain specification such as a test suite.

However, automated program repair systems suffer from the so-called *weak test* problem [37], which means that test suites are usually insufficient. Even if a patched program passes all tests, it may still be incorrect. In fact, many current program repair systems produce a large number of incorrect patches [30], such that developers cannot accept the generated patches blindly, and reviewing the patches is inevitable. Besides, current approaches only provide patches without any explanation, which makes the comprehension of patches more difficult for code reviewers. For example, Tao et al. [45] have shown that patches without enough communications to developers are likely to be rejected in open source projects.

Therefore, a more desirable approach for automated program repair is to generate an explanation along with a patch to facilitate the patch review. There are some researchers who have claimed the importance of explaining patches. For example, Le Goues et al. [17] believe that explaining repair is a strongly related problem when integrating repair tools into the development process. Monperrus [33] declares that a program repair must not only synthesize a patch but also synthesize the explanation. To synthesize the patch explanation, we first need to know how a patch should be explained, i.e., what elements constitute a patch explanation, and how they are expressed. As far as we are aware, no systematic study has been conducted to address this problem.

In this paper, we address the problem of *what kind of information was included in a well received patch explanation* by studying how developers explain patches in open source projects. More specifically, we aim to answer the following three research questions.

- **RQ1: What are the constituent elements for a patch explanation and how are they expressed?**
- **RQ2: What is the distribution of different constituent elements for patch explanations?**
- **RQ3: What is the distribution of different means for explaining constituent elements?**

RQ1 helps us to understand "what to explain" and "how to explain". The other two RQs help us understand how often the elements and means of explanation are presented and how they are correlated.

To answer these research questions, we collected 300 bug-fixing pull requests from six Java open-source projects on GitHub. All these pull requests have been merged, indicating that the explanations in these pull requests are more or less adequate. We manually analyzed all these pull requests to build a patch explanation model to address RQ1, and then we performed quantitative analysis over the elements and means of explanation in the model to answer RQ2 and RQ3.

Our study leads to a number of findings, and we highlight a few here.

This work is supported by the National Key Research and Development Program of China under Grant No. 2017YFB1001803, and the National Natural Science Foundation of China under Grant Nos. 61672045 and 61529201. Yingfei Xiong is the corresponding author.

978-1-7281-4982-0/19/$31.00 ©2019 IEEE
DOI 10.1109/ISSRE.2019.00016

- There are five core elements in a patch explanation, namely condition, consequence, position, cause, and change, each explained in different ways (called *expressive forms*).
- An explanation usually contains only two or three elements, indicating that we need to decide which element to present.
- Some elements have positive correlations (appearing together) while some have negative correlations (mutually exclusive).
- Reviewers are often assumed to be familiar with the project, and thus, instead of code terms, high-level abstract terms about the project are often used.
- While not popular among fault localization/predication approaches, variables are often used to point out the position of the bug in a patch explanation.
- Conditions are often specified over event sequence, which is not well supported in mainstream programming languages.

Our findings present a general understanding of patch explanations written by developers. These findings not only provide guidance for the research of patch explanation generation, but also are useful in other directions that require an understanding of patch explanations, such as guiding developers for writing better patch explanations, or measuring the quality of patch explanations.

The rest of the paper is organized as follows. Section II introduces our dataset. Section III presents our explanation model. Sections IV and V present our quantitative analysis. Section VI describes the implications of our study. Section VII discusses the threats to validity in our study. Section VIII presents the related work, and Section IX concludes our study.

II. Data Collection

A. Projects

In this study, we used six popular open-source Java projects on GitHub, whose basic information is shown in Table I. We selected these projects by considering various differences among them so as to increase the diversity of our data sample. As shown in Table I, the six projects are from different development organizations, cover both libraries and applications, and cover both sequential and concurrent projects. More specifically, *RxJava* is a library for composing asynchronous and event-based programs and is often used for building Android applications [1]. *Spring* is a framework for creating Java web applications [2]. *PocketHub* is an Android client for GitHub [3]. *Nextcloud* is an Android client for file share and communication [4]. *IntelliJ* is a widely-used Java IDE [5]. *Lang* is a library for Java utility classes [6].

TABLE I
Basic Information of Projects

Orgnization/Project	Domain	#Star	#PR	SLOC
ReactiveX/**RxJava**	lib	35,688	51	267.2K
Spring/**Spring**-Boot	frame	29,646	56	243.3K
PocketHub/**PocketHub**	app	9,311	40	157.3K
JetBrains/**IntelliJ**-Community	app	6,647	64	3,544.7K
Apache/Commons-**Lang**	lib	1,450	36	761.8K
Nextcloud/Android	app	1,090	53	59.4K

† "SLOC" refers to the number of lines of Java code, calculated by cloc [7]. "#PR" refers to the number of reviewed pull requests.

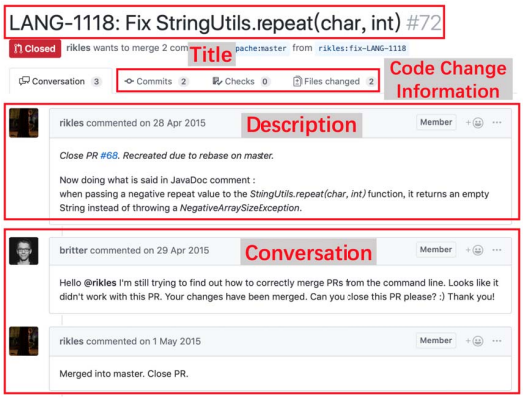

Fig. 1. An example of a pull request [Lang#72] on GitHub

B. Pull Requests

Figure 1 shows a screen shot of a typical pull request from the project *Commons-Lang* in GitHub, which consists of the following four parts:

- *Title* briefly summarizes the pull request.
- *Description* explains the pull request in detail, e.g., what bugs the pull request fixes and how to fix the bugs.
- *Conversation* shows the discussion between the submitter and reviewers.
- *Code Change Information* shows the commits, changed files, and code change difference, whose details can be found in the corresponding tags.

In our study, we collected all the pull requests before Jan. 21th, 2019 for each project. We then applied the following criteria to include pull requests from the initial collection:

The pull request has been merged to the master branch. We only focus on merged pull requests because merged pull requests have been reviewed as valid, and its explanation is probably adequate for the developers to understand the change. In our study, we follow Gousios' heuristics [18] to detect merged pull requests, including integrated through other mechanisms and GitHub's user interface.

The pull request changes Java source code. Even if the main programming language of a project is Java, the changed code in the pull request may be other programming languages, such as XML or Markdown. In our study, we focus on bugs in Java programs, and thus we considered only the pull requests that mainly changed Java source code, excluding test code. We checked this condition by examining the concrete changes.

The pull request only involves bug-fixing. A pull request may have different purposes, e.g., adding new features, refactoring, or bug-fixing. In this study, we only focus on the bug-fixing pull requests. We applied the following criteria to decide

whether a pull request fixes bugs: (1) the pull request has been labeled with "Bug"; or (2) the link of a bug report labeled with "Bug" is provided in the description or conversation. To guarantee that the pull request only fixes bugs, we select the pull request involving the only one label: "Bug".

A possible alternative is to also review unmerged requests, and treat these requests as negative examples to understand the difference between good and bad explanations. However, we rejected this idea because the reasons of rejections are various. A unmerged requests may also be well explained, but was rejected by other reasons such as incorrect code.

For every project, we randomly selected pull requests that match the criteria mentioned above one by one until our analyzed results achieve saturation. The pull requests, whose descriptions are only the words "Commit" or "Bug fix", were excluded. In the end, we totally reviewed 300 pull requests. Our experimental data can be found at *https://github.com/anonymity2019/issre.*

III. RQ1: What are the constituent elements for a patch explanation and how they are expressed?

A. Methodology

We use open coding [25] on all the sampled pull requests to first generate the element types and then manually label pull requests with the derived element types. The detailed process is presented as follows: First, two authors individually coded the sampled data set by reading the title, description, and conversation, and summarized the elements for each pull request; during a second pass, the codes were extracted. The extracted codes were then grouped together and processed to remove duplicates and, in cases, to generalize or specialize them. The new codes were then applied to all answers. When new codes emerged, they were integrated in the code set. During the process, the code change information is used to better understand the pull request and verify the coding results. The output of this step is the *elements* a patch explanation contains, and the relationship among these elements. Once an inconsistency between the two authors was produced, it was discussed with other authors until a consensus was reached.

After identifying all the elements, we follow the same process to further subdivide each element into *expressive forms* in order to capture different ways to express an element. Finally, we built a patch explanation model.

B. Patch Explanation Model

Figure 2 shows the patch explanation model we defined, which specifies how a patch explanation is formed. In the model, each line indicates that the item on the left is composed of the items on the right, where items on the right are optional, but at least one of them should be presented per each element.

According to our analysis result, a patch explanation consists of five elements (Line 1): `Condition`, `Consequence`, `Cause`, `Change`, and `Position`, and Table II summarizes each element, and corresponding information.

`Condition`, `Consequence`, and `Cause` describe information about the bug, and `Change` describes information related to the repair. `Position` involves both bug and repair information. `Condition` is the condition to trigger the bug, `Consequence` is the undesirable consequence that the bug leads to, `Position` is the location in the source code that causes the bug and is repaired, `Cause` is the reason why the bug occurs at the location, and `Change` is the modification applied to the source code to fix the bug. In other words, these elements explain *when*, *where*, and *why* the bug occurs, *what* it causes, and *how* it is repaired.

Also, `Position`, `Cause` and `Change` concern about the static, syntactic parts of the program, and `Condition` and `Consequence` concern about the dynamic, runtime behaviors of the program.

Each element could be expanded into expressive forms (Line 2-6), and some expressive forms can further be expanded. We define the expressive forms that could be further expanded as *intermediate expressive forms* (marked with [ ]), otherwise, they are *leaf expressive forms* (marked with <>). We shall only concern leaf expressive forms in the quantitative studies later, and shall refer *leaf expressive forms* directly as expressive forms if no confusion would be caused.

Figure 3 shows a patch explanation[1], which contains one `Condition` element and two `Consequence` elements. The element `Condition` describes a dynamic program `state`, where the function's argument is a negative repeat value. One `Consequence` element expresses the `expected` program `state` ("*it returns an empty String*") and the other `Consequence` element expresses the `actual occurred_event` ("*throwing a NegativeArraySizeException*").

TABLE II
Characteristics of the core elements

Element	Explanation	Info_Category	Statics/Dynamics
Condition	When the bug occurs	Bug	Dynamics
Consequence	What the bug causes	Bug	Dynamics
Position	Where the bug occurs	Bug,Repair	Statics
Cause	Why the bug occurs	Bug	Statics
Change	How the bug is repaired	Repair	Statics

```
1.  EXPLANATION  ::=  Condition, Consequence, Position, Cause, Change
2.  Condition    ::=  [dynamics]
3.  Consequence  ::=  [expected], [actual]
4.  Position     ::=  <file>, <inner_class>,
                      <method>, <variable>, <module>
5.  Cause        ::=  <missing_process>, <wrong_process>
6.  Change       ::=  <insertion>, <deletion>,
                      <replacement>
7.  [expected]   ::=  [dynamics]
8.  [actual]     ::=  [dynamics]
9.  [dynamics]   ::=  <state>, [event]
10. [event]      ::=  <missing_event>, <occurred_event>
```

Fig. 2. Patch Explanation Model
[ ] → Intermediate Expressive Form, <>→ Leaf Expressive Form

C. Elements and Expressive Forms

Now we introduce the elements and their expressive forms defined in the model.

[1]https://github.com/apache/commons-lang/pull/72.

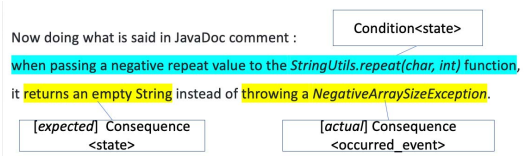

Fig. 3. An example [Lang#72] of expressive forms compose to elements

1) Condition: `Condition` expresses a condition in the runtime execution where the buggy behavior would be triggered. Thus `Condition` could be expressed by `dynamics`, which could be further expanded to `state` or `event`, indicating that a dynamic condition can be expressed either by a condition on the state of the program or by a condition over the sequence of events happened during execution. Similarly, `event` could be further expanded to `missing_event` and `occurred_event`, indicating that the condition over the sequence of events can be expressed by the missing or existences of events in the sequence.

a) Describing condition over a state: When describing a condition over a state, the typical pattern is to declare a predicate over the variables at the corresponding position. The most common predicate is that a variable is equal to a certain value or has a specific type. For example, Example 1 shows a condition where the *second argument* is not a typical (*Wildcard*) type. A predicate can be more general and captures a range of states. For example, Example 2 shows a condition where the input value contains lower cases.

Example 1: *"fixes bug in TypeUtils.equals(WildcardType, Type) where it was incorrectly returning true when* ***the second argument was not a Wildcard type****."* [Lang#73]

Example 2: *" Currently, IllegalArgumentException occur* ***if contains lowercase into log level.****"* [Spring#7914]

Besides *concrete conditions* like the above that correspond to concrete code elements, a condition can also be *abstract*, typically by declaring conditions over abstract concepts that are known among the developers. For example, Example 3 shows an abstract condition requiring the presence of multiple datasources, where "datasource" is an abstract concept known among developers.

Example 3: *"****When there are multiple datasources present,*** *make sure that the AutoConfigureTestDatabase annotation marks the embedded source as primary."* [Spring#7217]

b) Describing condition over a sequence of events: Besides describing a condition over the state, the other expressive form is to describe a condition over the sequence of events occurred during the program execution. An event may refer to concrete events, such as calling a method or throwing an exception, abstract events such as opening a file, or user events such as clicking a button. Example 4 shows an abstract event of setting `management.port` where the actual implementation code involves creating an object and calling its methods. Example 5 shows a user event, typically occurred in applications with GUI.

Example 4: *"Webflux doesn't require Servlet.class,* ***when setting management.port****, auto-configuration would fail with class not found exception."* [Spring#10590]

Example 5: *"****Click one issue of "New" tab in Home page. Click home/back button on the toolbar in the opened issue page.*** *Will see a strange loading view on the toolbar."* [PocketHub#1082]

Please note that in theory all assertions over the event sequence can be declared over states, as the events must have caused some changes in the state to trigger the bug. In Example 5, the events that click the tab and button must cause some values of program variable to be changed. However, developers still use many assertions (61.1%) over event sequences, suggesting such assertions may be more natural to the developers. While we can easily declare assertions over states in current programming languages, it is not easy to declare assertions over event sequences. Our results suggest that such a mechanism may be convenient to the developers if presented at the programming language level.

> **Observation 1:** *When describing runtime conditions, either assertions over the current program state or assertions over the event sequence may be used.*

2) Consequence: `Consequence` captures what unexpected result the bug caused. Similar to `Condition`, `Consequence` also describes that undesirable behavior happens at runtime and can be described from two aspects: the `expected` behavior, or the `actual` undesired behavior. For example, Example 6 explains the expected behavior, where *Follow* should be hidden. Example 7 explains the actual behavior, where an undesirable exception *IllegalArgumentException* is thrown.

Example 6: *"****Hides 'Follow' on*** *if the viewed user is the current user."* [PocketHub#955]

Example 7: *"Currently,* ***IllegalArgumentException occurs*** *if contains lowercase into log level."* [Spring#7914]

The consequence for the expected behavior and for the actual behavior can often be deduced from each other. Thus, it is common to have one of them in a patch explanation. However, this is not always the case. For example, Example 8 describes both the expected and actual results, because the two results cannot be deduced from the other, and both provide useful information of the bug.

Example 8: *" The following scenario did not work...* ***expected result: result stream emits the combined event actual result: result stream does not emit anything"*** [RxJava#5494]

> **Observation 2:** *When explaining the consequence of a bug, the expected behavior and the actual behavior can often be mutually deduced from each other in most cases, and 87.2% of the consequence elements mention only one of the them.*

The same as `Condition`, the expected and actual consequences are both expressed by `dynamics`, which expands to either state or event. Example 9 shows a consequence state *'true'*, example 10 shows the `occurred_event` in StackOverflowError, and Example 11 contains the `missing_event` "does not throw an exception" in the program.

Example 9: *"fixes bug in TypeUtils.equals(WildcardType, Type) where it was incorrectly returning* ***true*** *when the second argument was not a Wildcard type."* [Lang#73]

Example 10: *"****StackOverflowError*** *on TypeUtils.toString (...) for a generic return type of Enum.valueOf."* [Lang#292]

Example 11: *"ReflectionToStringBuilder* ***doesn't throw IllegalArgumentException*** *when the constructor's object param is null."* [Lang#85]

3) Position: `Position` explains the position in the source code where the bug occurs and is repaired. `Position` can be specified concretely using source code in the code, or can be specified abstractly using high level terms. Our model, `Position` expands to five expressive forms: `file`, `inner_class`, `method`, `variable`, and `module`. The first four expressive forms are concrete, referring to elements in the code. For example, in Example 12, *StrBuilder* is a file name and *replaceImpl* is a method name. The expressive forms do not include `class` because in Java the file name is required to be the same with the main class name. The expressive forms also do not include statements or line numbers, and we observe the reason is that these information can be easily seen from the concrete changes and the submitter does not need to mention them again. Furthermore, as Example 12 shows, an expressive form with a smaller code granularity does not necessarily subsume an expressive form with a larger code granularity. For example, knowing method *replaceImpl* does not necessarily mean that we know the file *StrBuilder*, as methods with this name can appear in different classes.

Example 12: *"Fix issue of buf using nonupdated buffer in* ***StrBuilder replaceImpl****. Avoid array OoB error by keeping variable buf consistent with buffer."* [Lang#200]

An interesting observation here is that, when submitters refer to code elements, they seldom explicitly point out the type of the code elements. In Example 12, the submitter does not explicitly mention that *StrBuilder* is a file while *replaceImpl* is a method. This observation suggests that the submitter usually assume reviewers are familiar with the code, an assumption that can be taken into account when designing explanation generation approaches.

> **Observation 3:** *The submitters of pull requests do not explicitly mention the type of code element, assuming the reviewers are familiar with the code and can easily identify them.*

Another interesting observation is that variables are used to explain the position of a bug in some pull requests. For example, Example 13 shows a mentioned position which is a concrete variable *swappedPair*. In many studies of fault localization [24], [55] and fault prediction [12], bugs are usually localized or predicted at statement, method, class, or file level. This finding suggests that variable may also be a good way to express the location of a bug and should be taken into account.

Example 13: *"There seems to be a bug in the current implementation of the isRegistered implementation, where the* ***swappedPair*** *is constructed similarly to the existing pair to check their existence in registry."* [Lang#282]

> **Observation 4:** *Variables are used to represent the position of a bug compared to automatic fault localization technique.*

The last expressive form `module` captures a function related position. In many projects, developers often implicitly divide the source code into different modules to achieve different functions, and such module definitions are mutually understood among submitters and the reviewers. For example, the project *RxJava* implements a set of operators, and for each operator, a set of methods are implemented following a certain pattern. As a result, by only mentioning the name of an operator, the reviewer could easily locate the bug location. Example 14 demonstrates it.

Example 14: *"This PR fixes a deadlock issue with the* ***refCount operator*** *when ..."* [RxJava#5975]

A further observation is that, while the positions mentioned in pull requests are always modified by the patch, not all positions modified by the patch are mentioned in the pull requests. When a patch modifies multiple positions, usually only one position is mentioned by the submitter to highlight the most important and representative position. This observation suggests an approach highlighting the most important change may be useful to the user.

> **Observation 5:** *Among the explanations of patches that change multiple positions, 73.8% explanations mentions only one position.*

4) Cause: `Cause` explains why the bug occurs at a certain position of the program. `Cause` has two expressive forms, `missing_process` and `wrong_process`.

The expressive form `missing_process` captures the case where developers omitted some processing code. One possibility is the omission of a subset of input domain, i.e., developers forget to deal with some corner cases. Example 15 describes the program lacks a check for a subscriber. Another possibility is the omission of a procedure. Example 16 shows that the developer forgets to clear the cookies.

Example 15: *"Previously SingleFromCallable* ***did not check if the subscriber was unsubscribed*** *before emitting onSuccess or onError."* [RxJava#5743]

Example 16: *"Logout* ***never cleared the WebViews cookies*** *so you could not switch you accounts, we also need to clear the cached items in the database."* [PocketHub#1109]

The expressive form `wrong_process` captures the case where the current processing code is wrong. Example 17 describes the problem that the program does not properly handle the exclusion of two actions.

Example 17: *"The logic* ***didn't properly mutually exclude the timer action and the onNext action****, resulting in probabilistic emission of the same buffer twice."* [RxJava#5427]

5) Change: `Change` describes the modifications applied to the code to repair the bug. As the standard way to express modifications, `Change` has three expressive forms: `insertion`, `replacement`, and `deletion`. However, since the concrete changes are already given by the pull requests, the submitter always provides a high-level abstract summary of the change instead of a description of the concrete syntax differences. Example 18 explains a removal operation. However, instead of describing concrete positions of the removal, the submitter explains the conditions for applying the removal to show the positions abstractly. Example 19 is an example of insertion, where *"close connection"* is used to summarize the concrete statement inserted and "after performing the actual check" is used to abstractly specify the insertion location.

Example 18: *"This pull request removes the started check on stop() for Jetty and Tomcat."* [Spring#8227]

Example 19: *"Close connection after performing the actual check to release resources."* [Spring#10153]

> **Observation 6:** *In 86.4% of the pull requests describing the code change, code change is described in abstract ways to summarize the concrete changes that are already presented in the file changes.*

D. Inter-Rater Agreement

The two authors who participated in open coding respectively had 7 and 8 years of programming experiences. One was a student in computer department, while another was a software developer in industrial company. Both of them had rich experience for bug fixing.

Most of results labelled by two authors were consistent, but there were inconsistencies in both the elements and the expressive forms. When labeling elements, two authors had disagreement in 34/300 pull requests. `Cause` and `Consequence` caused the most inconsistencies, because the two elements can sometimes be inferred from each other and thus cause confusion. When labeling expressive forms, the two authors had disagreement in 26/300 pull requests. The expressive forms `event` and `state` caused the most inconsistencies, because assertions over the event can also be translated to those declared over states, causing confusion.

IV. RQ2: Elements Distribution in an Explanation

A. Methodology

In particular, we conduct quantitative analysis for RQ2 and RQ3 from the following three aspects.

- **A1**: How many elements a pull request usually contains?
- **A2**: What elements are more popular when explaining patches?
- **A3**: Is there a correlation between any two elements ?

To answer A1 of RQ2, we manually identified whether each of the five elements exists in each pull request of our dataset, and then counted the number of elements presented in each pull request.

To answer A2 of RQ2, we counted the number of pull requests containing each element, and calculated their proportions among all the pull requests.

To answer A3 of RQ3, we used a statistical metric called *lift* [8], which measures how much the occurrence of an item A increases the probability of the occurrence of an item B, as defined in Formula 1. In this formula, *per(A)* is the percentage of pull requests containing element A in all the pull requests, and *per(A,B)* is the percentage of pull requests containing elements A and B at the same time in all the pull requests. If *lift(A,B)* is equal to 1, i.e., per(A,B) = per(A) * per(B), elements A and B are not correlated. If it is greater than 1, elements A and B are more likely to appear in the same pull request. If it is less than 1, the two elements are less likely to appear in the same pull request.

$$\text{lift(A,B)} = \frac{\text{per(A,B)}}{\text{per(A)} * \text{per(B)}} \tag{1}$$

B. Results and Analysis

1) Number of elements involved in an explanation: Table III shows the percentage of pull requests containing the corresponding number of elements (#1–5). Overall, only 2.0% (i.e., six) pull request involves all of the five elements, while almost 70% pull requests involve two or three elements. For ease of presentation, we call them two-element and three-element pull requests.

TABLE III
PERCENTAGE OF PULL REQUESTS CONTAINING #1–5 ELEMENTS

Num.	RxJava	Spring	PocketHub	IntelliJ	Lang	Nextcloud	Total
#1	7.8%	19.6%	32.5%	18.8%	8.3%	30.8%	20.4%
#2	29.4%	42.9%	42.5%	45.3%	50.0%	63.5%	45.5%
#3	47.1%	23.2%	17.5%	23.4%	36.1%	3.8%	24.1%
#4	11.8%	12.5%	7.5%	7.8%	5.6%	1.9%	8.0%
#5	3.9%	1.8%	–	4.7%	–	–	2.0%

† 'Num' is the number of elements in a pull request.

It is reasonable to use two or three elements to explain a patch. First, it is scarcely possible for some elements to be used individually such as `Position` and `Condition`. Therefore, the percentage of one-element pull requests is relatively small. Second, it is unnecessary to use too many elements to explain a patch. There are two possible reasons. First, different elements may convey overlapped information.

For example, in Example 15, the cause of the bug is that the developer forgot a check for the subscriber, and a natural change for repairing this bug is to add the check. The elements `Cause` and `Change` are overlapped, and thus the element `Change` is omitted in the explanation. Second, some elements may be not critical to understand the patch. For example, Example 19 does not specify the condition under which the bug may occur, but such information may not be critical in understanding the bug. This observation suggests that, when generating explanation, it may be important to select which elements should be involved in the explanation.

When we consider each individual project, the observation is similar. That is, for three projects, the numbers of two-element and three-element pull requests are still the largest in each project, i.e., *RxJava* 76.5%, *Spring* 66.1%, and *Lang* 86.1%. The results are slightly differet for *PoketHub*, *Nextcloud* and *IntelliJ*, where the percentages of one-element pull requests are larger than that of three-element pull requests. We further analyzed this issue, and found that the one-element pull requests in *PoketHub*, *Nextcloud* and *IntelliJ* mainly contain `Consequence`. A possible reason for their percentage difference is that their domains are different. *PoketHub*,*Nextcloud* and *IntelliJ* are application projects, which may require less explanation than library and framework projects. In many cases, a single expected consequence describes what happened after applying the patch, which is enough to explaining the purpose of a patch for application projects. For example, Example 20 only describes an expected consequence that is after applying the modification the system could correctly displaying characters.

Example 20: *"modified to display characters of some languages(eg. chinese) correctly"* [PocketHub#466]

Observation 7: *Almost 70% pull requests contain two or three elements.*

We also analyze that, for all projects, most one-element pull requests use `Change` or `Consequence` alone.

That is, `Change` or `Consequence` alone could well describe patches to some degree. For two-element pull requests, `Position` occupies a significant portion. That is, while it is not effective when used alone, `Position` is helpful when combined with other elements.

Observation 8: `Change` *or* `Consequence` *alone could describe many patches, while* `Position` *is more often used together with other elements.*

2) Popular Elements: Table IV shows the percentage of pull requests containing each type of elements. As shown in this table, the most frequently used element is `Consequence`(75.7%). This is as expected, because most of the submitters have encountered the consequences of the bug themselves, and thus it is natural to express `Consequence`. The second frequently used element is `Condition` (48.3%). This is because bugs are triggered under certain `Condition`, and thus when describing `Consequence`, submitters usually mention `Condition`. `Cause` (22.3%) has the smallest percentage. This is because developers have domain knowledge for each project and can deducted `Cause` when `Change` is explained. For example, Example 21 only describes how did the submitter fix the bug without explaining the cause, but it is easy for developers to realize that the cause of bug is missing null check.

Example 21: *"DiffBuilder: Add null check on fieldName when appending Object or Object[]."* [Lang#121]

TABLE IV
PERCENTAGE OF PULL REQUESTS CONTAINING EACH TYPE OF ELEMENTS

Element	RxJava	Spring	PocketHub	IntelliJ	Lang	Nextcloud	Total
Position	76.5%	48.2%	12.5%	37.5%	88.9%	3.8%	31.7%
Condition	51.0%	41.1%	57.5%	53.1%	36.1%	49.1%	48.3%
Consequence	76.5%	60.7%	80.0%	81.3%	50.0%	98.1%	75.7%
Cause	27.5%	23.2%	30.0%	21.9%	19.4%	13.2%	22.3%
Change	43.3%	55.4%	25.0%	42.2%	44.4%	9.4%	37.0%

Observation 9: `Consequence` *and* `Condition` *are the two most popular elements, while* `Cause` *is the least.*

For each project, the conclusions are also similar. For example, `Consequence` is indeed frequently used in each project, which is the second most popular element for *Lang* and the most popular element for all the other projects. However, there are also some differences for different projects. For example, the pull requests of framework and library projects use more `Postion` than `Condition`, while those of the Android application rarely use `Position`. Through our manual investigation, one possible reason is that about 40% pull requests of *PocketHub* and *Nextcloud* are described in terms of UI interactions, which is hard to be directly mapped to a position in the source code.

3) Correlation of Elements: For A3, we computed the correlation of any two elements, whose results are shown in Table V. The bold numbers represent the corresponding elements that have the most strongly positive correlation, and the italic numbers represent the elements have the most strongly negative correlation.

TABLE V
THE CORRELATION BETWEEN TWO ELEMENTS

	Position	Condition	Consequence	Cause	Change
Position	–	–	–	–	–
Condition	0.91	–	–	–	–
Consequence	0.79	1.11	–	–	–
Cause	1.15	0.80	0.80	–	–
Change	**1.17**	0.71	*0.61*	0.97	–

The pairs with the two most positive correlations involve `Position`. The reason is that, when explaining a cause and change, `Position` is often used as auxiliary information and its existence is dependent of other elements. There are another pair whose correlation is positive: `Condition` and

`Consequence`. This is as expected: a single condition without consequence makes no sense.

On the other hand, both `Condition` and `Consequence` are negatively correlated with the other elements except each other. Especially, `Consequence` and `Change` are strongly negatively correlated. We explored the reason and found that many pull requests are the fixes of simple bugs, where reading `Condition` and `Consequence` is enough for developers to deduce other information about bugs.

Observation 10: `Position` *has the most positive correlation with* `Cause` *and* `Change`*. Both* `Condition` *and* `Consequence` *are negatively correlated with the other elements except each other.*

V. RQ3: Expressive Forms Distribution in an Explanation

A. Methodology

The methodology of RQ3 is similar to RQ2, except that we applied the analysis to expressive forms instead of elements from A1 and A2. We did not conduct experiment for A3 because the data quantity is not enough to support correlation analysis.

B. Results and Analysis

1) Number of Expressive Forms per Element: First of all, we found that some elements have mutual exclusive expressive forms and involve at most one expressive form in a pull request, such as `Condition`, `Cause`, and `Change`. Since abstract description can be used, with proper abstraction, all the bugs can be triggered with a single condition, have one single cause, and require one operation to repair. Therefore, these elements in our dataset do not require multiple expressive forms to describe.

For `Position` and `Consequence`, their expressive forms are complementary, Table VI and Table VII show the percentages of the elements `Position` and `Consequence` using the corresponding number of expressive forms to express, respectively.

TABLE VI
PERCENTAGE OF `Position` USING #1–3 EXPRESSIVE FORMS

Num.	RxJava	Spring	PocketHub	IntelliJ	Lang	Nextcloud	Total
#1	71.8%	51.9%	100.0	62.5%	18.8%	50.0%	53.5%
#2	28.2%	33.3%	–	33.3%	78.1%	50.0%	41.9%
#3	–	14.8%	–	4.2%	3.1%	–	4.7%

† 'Num' is the number of expressive forms in position. Position expressive forms including file, inner_class, method, variable and module. #1 means a position only refers one of above five expressive forms.

Multiple expressive forms are used for `Position` since one expressive form often cannot pinpoint a precise location. We found submitters could use up to three expressive forms in one pull request in our dataset. As we can see from Table VI, most pull requests contain one expressive form (53.5%) or two expressive forms (41.9%), with a few containing three (4.7%). There are also some variations between projects. First, the position information for *PocketHub* pull request is simple, which only contains one expressive form. The reason is that, since *PocketHub* is a GUI application, most pull requests describe the bug using user actions, rather than pinpoint the position in the code. Second, *Spring* has much more expressive forms than other projects. A possible reason is, as a framework, *Spring* usually requires precise discussion of the code.

TABLE VII
PERCENTAGE OF `Consequence` USING #1–2 EXPRESSIVE FORMS

Num.	RxJava	Spring	PocketHub	IntelliJ	Lang	Nextcloud	Total
#1	89.7%	88.2%	84.4%	88.5%	77.8%	88.5%	87.2%
#2	10.3%	11.0%	15.6%	11.5%	22.2%	11.5%	12.8%

† 'Num' is the number of expressive forms in consequence. Only consider two kinds of expressive forms: actual and expected. #1 means a consequence only refers actual or expected.

`Consequence` has multiple expressive forms because some pull requests describe both expected behaviors and actual behaviors, as Example 8 shows. From Table VII, we can find that about 87.2% pull requests only mention one expressive form, while 12.8% pull requests mention both actual and expected results. When considering each individual project, the situation is almost the same: the percentage of pull requests mentioning one kind of consequence is much larger than the percentage of pull requests mentioning two kinds of consequence.

Observation 11: *In most cases elements are described with one expressive form, but* `Position` *may require multiple expressive forms to pinpoint a precise location, and* `Consequence` *may be expressed from both expected and actual behaviors.*

2) Popular Expressive Forms: Table VIII shows the proportion of expressive forms for each element. We will discuss each element one by one.

`Position`. As shown in Table VIII, 38.0% of the `Position` elements involve the file name, which is the most common position information. 37.2% of the `Position` elements involve the method name. More concrete positions such as variables are less frequently used. This observation is consistent with existing studies [27] that developers expect fault localization approaches to working on the method level.

The expressive form `module` is only used in *RxJava*, since only this project has a clear notion of module that is mutually understood among submitters and reviewers.

Observation 12: *Coarse-grained positions such as files and methods are more frequently used than fine-grained position such as variables.*

`Condition`. From the Table VIII we can see that, `missing_event` is never used for conditions in our dataset. This observation implies that fewer bugs are related to the

absence of events than the presence. In addition, `state` is used less frequently than `occurred_event`, indicating that an event is more important than a state to express a condition.

`Consequence`. Consequence can be described from the actual behavior or the expected behavior. As we can see from the table, submitters are more likely to describe the undesirable actual behavior rather than the expected behavior. Also, events are much more frequently used than states. When `event` is used, whether the consequence is described through missing events or occurred events depending on the nature of the bug. As we can see from the table, each type occupies a significant portion. Please note here a missing event in `actual` corresponds an occurred event in `expected`, and vice versa.

> **Observation 13:** *While conditions are described through both events and states, consequences are much more often described through events.*

`Cause`. How `Cause` is expressed depends on the bug type rather than the choice of submitters. Generally, a bug occurrence can be divided into two case: one is wrong handling case, another is missing handling case. Table VIII shows that the proportion of two kinds of expressive form are: `wrong_process` 74.6% and `missing_process` 25.4%.

`Change`. Similar to `Cause`, how `Change` is expressed also depends on the bug type. As shown in Table VIII, the percentage of three expressive forms are: insert (36.7%), delete (5.1%), and replace (58.2%). The results suggest that most bugs require addition and replacement to repair, while a few require deletion. This is consistent with several existing program repair approaches that treat deletion as an anti pattern [43].

From the table we can also observe project-specific characteristics of the expressive patterns. We have already mentioned that the pull requests of *Lang* use more expressive forms for positions than other projects. Here we can also observe that `method` is more frequently used in *Lang* than other projects. Furthermore, regarding to `Condition`, GUI applications (*PocketHub* and *Nextcloud*) and event-based software (*RxJava*) are more likely to be specified via events while libraries (*Lang*) are more likely to be specified via state. It is interesting that the application *IntelliJ* uses `state` more frequently than the other two applications (*PocketHub* and *Nextcloud*). We investigated the pull requests of *IntelliJ*, and found that the reason is mainly related to the complexity of *IntelliJ*. Since *IntelliJ* is a complex IDE, the state triggering the bug can often be reached via different event sequences, and it is easier to specify the condition via state. For example, several bugs are related to the conflicts between plugins, it is easier to specify the condition as "the existence of XX and XX plugins at the same time" rather than describing the plugin installation process.

> **Observation 14:** *The use of expressive forms depends on the category of the project and the type of bug.*

TABLE VIII
THE EXPRESSIVE FORM FREQUENCY OF ELEMENTS

		RxJava	Spring	PocketHub	IntelliJ	Lang	Nextcloud	Total
Pos	file	38.5%	44.4%	60.0%	58.3%	12.5%	50.0%	38.0%
	method	23.1%	22.2%	–	29.2%	78.1%	50.0%	37.2%
	inner_class	–	–	–	–	3.1%	–	0.6%
	variable	–	33.8%	40.0%	12.5%	6.3%	–	12.4%
	module	38.5%	–	–	–	–	–	11.6%
Cond	miss_e	–	–	–	–	–	–	–
	occur_e	80.8%	43.5%	81.8%	58.8%	–	73.1%	61.1%
	state	19.2%	56.5%	18.2%	41.2%	100.0%	26.9%	38.9%
Consq	state_actual	–	17.6%	25.0%	11.5%	27.8%	–	11.5%
	miss_e_actual	7.7%	14.7%	15.6%	9.6%	11.1%	–	8.8%
	occur_e_actual	71.8%	88.2%	31.3%	57.7%	61.1%	96.2%	70.0%
	state_expt	–	8.8%	21.9%	–	–	–	7.5%
	miss_e_expt	–	–	–	–	–	–	–
	occur_e_expt	30.8%	8.8%	21.9%	23.1%	–	15.4%	18.5%
Cause	miss_process	50.0%	30.8%	25.0%	14.3%	14.3%	–	25.4%
	wrong_process	50.0%	69.2%	75.0%	85.7%	85.7%	100.0%	74.6%
Change	insert	36.4%	33.3%	–	40.7%	50.0%	20.0%	36.7%
	delete	9.1%	11.1%	–	–	6.3%	–	5.1%
	replace	54.5%	55.6%	80.0%	59.3%	43.8%	–	58.2%

† 'e' is short for event. 'expt' is short for expected. Note that we only consider the expressive forms with the smallest granularity in a pull request. The reason why the sum is less than 100% is round-off errors.

VI. IMPLICATIONS

Our results have several implications. First of all, our results are useful for the generation of patch explanations. Our model has captured the basic elements that should be involved in a patch explanation and the expressive forms to represent them, formed a basis for designing explanation generation. Our quantitative study has revealed which elements and expressive forms are more frequently used, which elements appear together or are exclusive. Our findings also review some facts about the reviewers. For example, reviewers can be assumed to be familiar with the projects and high-level abstract terms are often preferred than code. These findings should be useful in patch generation.

While generating full explanation is hard, our findings also reveal parts of explanation that can be independently generated and could be useful to the reviewers. For example, one possibility is to point out the core modification if a patch modifies many places in the project. Such a task is much easier to automate than generating the whole explanation.

Several research efforts have been put on modeling bug report quality [20], and our results could help to extend these models for pull request quality. Similarly, many companies provide guidelines for bug report writers. Our results could further help extend such guidelines to pull requests.

Our results suggest a new granularity to be used in fault localization/prediction approaches: variable. Our findings also confirm that coarse-grained granularity such as methods in fault localization is preferred by the developers than fine-grained granularity such as statements [27].

Our finding that elements are often described in the abstract level confirms the usefulness of code summarization approaches [15], [31], and the collected pull requests provide data for the research in this direction.

One goal of programming language design is to let the developers specify the program in a natural way. Our results

found that conditions are often specified on event sequence but such a specification is not supported by mainstream programming languages, suggesting a new feature that could be considered by language designers.

VII. Threats to Validity

The *internal* threat to validity lies in our manual inspection of pull requests. To reduce the subjectiveness of the manual process, we adopted the open coding protocol, and two authors independently analyzed the dataset to obtain the elements and expressive forms until saturation is reached. We determine that the saturation is reached when the elements and expressive forms do not change after analyzing more than 20 pull requests.

Another *internal* threat to validity lies in the use of merged pull requests. We assume the merged pull requests are adequately explained, but a pull request may also be merged even if the explanation is insufficient. To mitigate this threat, we investigate 300 pull requests from 6 different projects. Furthermore, many of findings, such as what elements exist in patch explanations, are not affected by whether individual explanations are adequate or not. In addition, we found that half of the reviewed pull requests were submitted by the core developers of the projects. Actually, whether a pull request comes from the core developers has no effect on the model.

The *external* threat to validity lies in our dataset. Since each project tends to have its specific characteristics and its participants have specific expressions, our observations from our dataset may be not generalized to other projects well. To reduce this threat, we carefully constructed our dataset by selecting diverse projects that come from different organizations and different domains. However, our dataset only considers Java projects and is from GitHub. More studies are needed to understand whether the results are generalizable to other programming languages and repositories.

The *construct* threat to validity lies in the used correlation metric. We used *lift* as the metric to calculate the correlation between two elements. *Lift* may be not quite sufficient, and there are other metrics. However, these metrics may be not appropriate to our situation. For example, Pearson correlation coefficient [9] measures the linear correlation between two variables while elements in our model are not. We will try to use more metrics to confirm our results in future study.

VIII. Related Work

Empirical Studies. There are several empirical studies concern the communication process between developers and users around the pull requests, code changes, and bugs. First, researchers have investigated the factors that affect the pull request evaluation process [18], [19], [38], [40], [45]–[47], [51]. Results show that both technical and social factors affect the chance of acceptance. For example, programming languages and domain specific factors can influence the success and failure probabilities of pull requests. In particular, Tao et al. [45] investigated the rejected patches in Eclipse and Mozilla, and summarized the common reasons of patch rejections. Second, Tao et al. [44] consider the change understanding and analysis process of developers, and summarize the issues considered by the developers when understanding a (potentially undocumented) code change. Third, Bettenburg et al. [10] studied what makes a good bug report and revealed that steps to reproduce, stack traces, and test cases are helpful for developers to debug. Fourth, several approaches studies pull request triaging problem, that is, automatically recommending reviewers for pull requests [22], [28], [50], [52]–[54]. However, none of these studies answers the research questions of this paper: how patches are explained, what elements are mentioned in the explanations, and how they are expressed.

Automatic Code Comments Generation. Researchers have worked on automatically generating code comments in the literature. Most of these techniques mined existing projects, documentations, communications between developers to generate comments [14], [21], [26], [35], [36], [48], [49]. For example, ColCom [48] generates code comments by applying code clone detection techniques to discover similar code segments and used the comments from some code segments to describe the other similar code segments. Besides, Sridhara et al. leveraged the program analysis and natural language analysis to automatically generate descriptive summary comments for Java methods [41], Java classes [34], and parameter comments [42]. These approaches generate comments in general and do not consider the specific model of patch explanations. We believe our approach could potentially be combined with these approaches to build better patch explanations.

Automatic Change Summarization. Multiple approaches have been proposed to automatically generate summarizations for code changes [11], [13], [23], [39]. These approaches use text or syntactic differencing algorithms to identify and represent the changed part of the code, and extract comments or documentation related to the changed elements for developers' reference. In other words, these approaches mainly generate the concrete explanation for the `Change` element. Our studies reveal there are other elements such as `Condition`, `Consequence`, `Position`, and `Cause` often used in explaining patches, and `Change` element is often explained abstractly. These elements and expressive forms could be the future targets for summarization approaches.

IX. Conclusion

In this paper, we investigated the question "how to explain a patch", with a manual inspection on 300 bug-fixing pull requests from six projects in GitHub. We built a patch explanation model, revealing the core elements and their expressive forms in a patch explanation. We also performed quantitative analysis over the elements and patch explanations to understand their distributions and correlations. Our study leads to a set of findings which could be useful in different research problems, including generating patch explanation, measuring pull request quality, helping developers to create better pull requests, fault localization, fault prediction, code summarization, and programming language design.

REFERENCES

[1] https://github.com/ReactiveX/RxJava.
[2] https://github.com/spring-projects/spring-boot.
[3] https://github.com/pockethub/PocketHub.
[4] https://github.com/nextcloud/android.
[5] https://github.com/JetBrains/intellij-community.
[6] https://github.com/apache/commons-lang.
[7] https://github.com/AlDanial/cloc.
[8] https://en.wikipedia.org/wiki/Lift_(data_mining).
[9] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.
[10] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 308–318, 2008.
[11] R. P. L. Buse and W. Weimer. Automatically documenting program changes. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 33–42, 2010.
[12] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert Syst. Appl.*, 36(4):7346–7354, 2009.
[13] L. F. Cortes-Coy, M. L. Vásquez, J. Aponte, and D. Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, pages 275–284, 2014.
[14] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 47–57, 2012.
[15] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver. Evaluating source code summarization techniques: Replication and expansion. 33(2):13–22, 2013.
[16] A. Ghanbari and L. Zhang. Practical program repair via bytecode mutation. 2018.
[17] C. L. Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Commun. ACM*, 2019.
[18] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.
[19] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. Technical report, Delft University of Technology, Software Engineering Research Group, 2014.
[20] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE*, pages 34–43, 2007.
[21] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, and X. Luo. Mining version control system for automatically generating commit comment. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, pages 414–423, 2017.
[22] J. Jiang, Y. Yang, J. He, X. Blanc, and L. Zhang. Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull-based development. *Information & Software Technology*, 84:48–62, 2017.
[23] S. Jiang and C. McMillan. Towards automatic generation of short summaries of commits. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 320–323, 2017.
[24] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 467–477, 2002.
[25] S. H. Khandkar. Open coding. *University of Calgary*, 23:2009, 2009.
[26] J. Kim, S. Lee, S. Hwang, and S. Kim. Enriching documents with examples: A corpus mining approach. *ACM Trans. Inf. Syst.*, 31(1):1:1–1:27, 2013.
[27] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 165–176, 2016.
[28] Z. Liao, Y. Li, D. He, J. Wu, Y. Zhang, and X. Fan. Topic-based integrator matching for pull request. In *2017 IEEE Global Communications Conference, GLOBECOM 2017, Singapore, December 4-8, 2017*, pages 1–6, 2017.
[29] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of applications software maintenance. *Commun. ACM*, 21(6):466–471, 1978.
[30] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.
[31] P. W. McBurney and C. McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2016.
[32] M. Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys*, 2017.
[33] M. Monperrus. Explainable software bot contributions: Case study of automated bug fixes. 2019.
[34] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, pages 23–32, 2013.
[35] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora. ARENA: an approach for the automated generation of release notes. *IEEE Trans. Software Eng.*, 43(2):106–127, 2017.
[36] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pages 63–72, 2012.
[37] Z. Qi, F. Long, S. Achour, and M. C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 24–36, 2015.
[38] M. M. Rahman and C. K. Roy. An insight into the pull requests of github. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 364–367, 2014.
[39] S. Rastkar and G. C. Murphy. Why did this code change? In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1193–1196, 2013.
[40] D. M. Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino. Acceptance factors of pull requests in open-source projects. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1541–1546, 2015.
[41] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 43–52, 2010.
[42] G. Sridhara, L. L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, pages 71–80, 2011.
[43] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 727–738, 2016.
[44] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 51, 2012.
[45] Y. Tao, D. Han, and S. Kim. Writing acceptable patches: An empirical study of open source project patches. In *IEEE International Conference on Software Maintenance and Evolution*, pages 271–280, 2013.
[46] J. Terrell, A. Kofink, J. Middleton, C. Rainear, E. R. Murphy-Hill, C. Parnin, and J. Stallings. Gender differences and bias in open source: pull request acceptance of women versus men. *PeerJ Computer Science*, 3:e111, 2017.
[47] J. Tsay, L. Dabbish, and J. Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th international conference on Software engineering*, pages 356–366. ACM, 2014.

[48] E. Wong, T. Liu, and L. Tan. Clocom: Mining existing source code for automatic comment generation. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 380–389, 2015.

[49] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 562–567, 2013.

[50] H. Ying, L. Chen, T. Liang, and J. Wu. Earec: leveraging expertise and authority for pull-request reviewer recommendation in github. In *Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering, CSI-SE@ICSE 2016, Austin, Texas, USA, May 16, 2016*, pages 29–35, 2016.

[51] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *Mining software repositories (MSR), 2015 IEEE/ACM 12th working conference on*, pages 367–371. IEEE, 2015.

[52] Y. Yu, H. Wang, G. Yin, and C. X. Ling. Reviewer recommender of pull-requests in github. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 609–612, 2014.

[53] Y. Yu, H. Wang, G. Yin, and C. X. Ling. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In *Asia-Pacific Software Engineering Conference*, pages 335–342, 2014.

[54] Y. Yu, H. Wang, G. Yin, and T. Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information & Software Technology*, 74:204–218, 2016.

[55] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 14–24, 2012.