An empirical study on downstream workarounds for cross-project bugs

Hui Ding   Wanwangying Ma   Lin Chen   Yuming Zhou   Baowen Xu
State Key Laboratory for Novel Software Technology
Nanjing University, China
dinghui85@gmail.com, wwyma@smail.nju.edu.cn, {lchen, zhouyuming, bwxu}@nju.edu.cn

***Abstract*—GitHub has fostered complicated and enormous software ecosystems, in which projects depend on and co-evolve with each other. An error in an upstream project may affect its downstream projects through inter-dependencies, forming cross-project bugs. Though the upstream developers should fix the bugs on their side, proposing a workaround, i.e., a temporary solution in the downstream project is a common practice for the downstream developers. In this study, we empirically investigated the characteristics of downstream workarounds in the scientific Python ecosystem. Combining the statistical comparisons and manual inspection, we have the following three main findings. First, in general, the workarounds and the corresponding upstream fixes are significantly different in code size and code structure. Second, there are three kinds of cross-project bugs that the downstream developers usually work around. Last, four types of common patterns are identified from the investigated workarounds. The findings of this study lead to better understanding of cross-project bugs and the practices of developers in software ecosystems.**

Keywords—GitHub ecosystems; cross-project bugs; workarounds; practices

I. Introduction

Benefiting from the social coding capabilities of GitHub, software development on GitHub has evolved beyond a single project into socio-technical ecosystems [1]. Projects rely on the infrastructure or functional components provided by other projects, forming complex inter-project dependencies. In this way, some bugs in the upstream projects may affect their downstream projects through the dependencies. This phenomenon was confirmed by Ma et al. [2]. In their study, they investigated cross-project correlated bugs, i.e., causally related bugs reported to different projects in scientific Python ecosystem on GitHub, focusing on how developers coordinate to triage and fix this kind of bugs.

In the context of cross-project bugs, it is no doubt that the upstream project where the bug roots should provide a radical cure. However, the affected downstream projects usually offer a workaround, i.e., a temporary solution locally to bypass the upstream error. Ma et al. posted a questionnaire in which they asked what the downstream developers usually did to deal with cross-project bugs. The result indicated that 89.3% of the respondents chose to propose a temporary workaround, which was proven to be the most common practice [2].

Workarounds are important in two folded [2]. First, it can be used to avoid the long-lasting impact of an upstream bug. A workaround must be implemented if the upstream team is not willing or able to fix the bug quickly, and it allows the downstream project to temporarily suppress the upstream bug. Second, adding a workaround for an upstream bug enables the downstream project to support buggy upstream version without affecting the end users. As many users may still use an old version of the upstream project, the downstream developers cannot rely on a fix in the next upstream release. Therefore, the downstream developers have to work around bugs regardless of whether they have been already fixed upstream.

Despite the wide use and importance of the workarounds for cross-project bugs, little work has paid attention on this issue. Studying the workaround will help to understand not only the fixing process of cross-project bugs, but also the coordination between projects in a software ecosystem. Therefore, we conduct this study to investigate the characteristics of the downstream workarounds in the context of cross-project bugs.

We base our study on scientific Python ecosystem on GitHub. For a cross-project bug, we refer to the patch injected into the buggy upstream project as the upstream fix, while the temporary solution provided for the affected downstream project as the downstream workaround. We make an investigation of the workarounds from three aspects. First, we compare the code size and design of the workarounds with those of the corresponding upstream fixes. Second, we inspect whether the cross-project bugs that were worked around in downstream projects have something in common. Third, we investigate whether software practitioners developed the workarounds in some common ways.

The main contributions of this study is as follows. First, we extract 60 downstream workarounds in the scientific Python ecosystem. Second, we identify three kinds of cross-project bugs that the downstream developers usually work around. Third, we summarize four common workaround patterns. Last, we provide several design requirements for the workaround supporting tools.

The rest of the paper is organized as follows. Section II describes related work. Section III presents our research methodology, and Section IV shows our empirical results. We propose further discussions on our findings in Section V, and

978-1-5386-3681-7/17 $31.00 © 2017 IEEE
DOI 10.1109/APSEC.2017.38

examine threats to validity in Section VI. Finally, Section VII concludes this paper.

II. Related Work

A. Cross-project Bugs

As the development of software ecosystems, more and more cross-project bugs appear and attract the attention of an increasing number of researchers.

Some existing studies showed that cross-project bugs brought many troubles to ecosystem developers. Decan et al. [3] reported that the developers in R ecosystems felt it more and more of a pain if the upstream packages broke. Adams et al. [4] indicated that the core activity of integration for open source distributions was synchronizing the newer upstream version. To avoid the cross-project bugs, developers had to pay great attention on the synchronizing process. Bavota et al. [5] found that the upstream upgrade would have strong effects on downstream projects when there were general dependencies between them. Their study showed that a large amount of downstream code had to be modified when the upstream project changed if the downstream project depended on the upstream framework or general services. In that case, the upstream bugs would leave a wide impact on the downstream projects.

Some other researches focused on the coordination between developers in different projects during fixing cross-project bugs. Villarroel et al. [6] leveraged the reviews of App users to help developers realize the downstream demand. They classified and prioritized the downstream reviews, so that the upstream developers were able to catch the important bugs quickly. Ma et al. [2] studied how developers fixed cross-project correlated bugs in scientific Python ecosystem. Combining manual inspection and the results of an online survey, they revealed how developers, especially those on the downstream side tracked the root cause of cross-project bugs and dealt with them to eliminate their bad effects. Our study bases on and extends that work. We focus on a specific but common practice of the downstream developers when facing cross-project bugs, i.e., proposing a workaround.

B. Blocking Bugs

Another special type of bugs is blocking bugs which are to some extent similar to cross-projects bugs. Blocking bugs prevent other bugs (in the same or other projects) from being fixes. It often happens because of a dependency relationship among software components. Under the environment, the developers cannot fix their bugs because the modules that they are fixing depend on other modules that have unresolved bugs. Due to their severe impact, some researchers have turned their eyes to blocking bugs.

Garcial and Shihab [7] found that it took two to three times longer to fix blocking bugs than non-blocked bugs. They then employed decision tress to predict whether a bug is a blocking bug or not. They extracted 14 kinds of features to construct the predictor and evaluated which features were most influential to indicate the blocking bugs.

Later, Xia et al. [8] proposed a novel method named ELBlocker to identify blocking bugs with the class imbalance phenomenon taken into account. ELIbloker utilized more features and combined multiple classifiers to learn an appropriate imbalance decision boundary. ELIBloker outperformed the method in [7] by up to 14.7% F-measure.

Unlike blocking bugs which prevent the fixing of bugs in the dependent modules, cross-projects bugs occur in upstream projects but affect the normal operation of the downstream projects. For the affected downstream modules/projects, the developers attempt to take some action to be released from the blocking/cross-project bugs in other components. In this paper, we investigate the downstream practices when facing cross-project bugs.

C. Design of Bug Fixes

Fixing software bugs is an important activity during software maintenance. Developers devote substantial efforts to design the bug fixes, which reflect the developers' expertise and experience. Various studies investigated the nature and design of bug fixes. Zhong and Su [9] extracted and analyzed more than 9000 real-world bug fixes from six Java projects. They obtained 15 findings which could gain insights on automatic program repair. Pan et al. [10] explored the underlying bug fix patterns and identified 27 bug fix patterns that were amenable to automatic detection. Park et al. [11] analyzed bugs which were fixed more than once to understand the characteristics of incomplete patches. They revealed that predicting supplementary patch was a difficult problem. Jiang et al. [12] conducted an study on the characteristics of Linux kernel patches that could explain patch acceptance and reviewing/integration time. Misirli et al. [13] proposed a measure to study the impact of fix-inducing changes. They found that the lines of code added, the number of developers who worked on a change, and the number of prior modifications on the files modified during a change were the best indicators of high-impact fix-inducing changes. Echeverria et al. [14] evaluated developers' performance on fixing bugs and propagating the fixes to other products in industrial Software Product Line.

According to different characteristics of bug fixes, researches developed various automatic tools to support bug repair. Goues et al. [13,14] used genetic programming to repair bugs in C programs, and evaluated what fraction of bugs could be repaired automatically. They generated a large, indicative benchmark set for systematic evaluations. Mechtaev et al. [17] presented a semantics-based repair method applicable for large-scale real-world software. Gu et al. [18] considered bad fix problem and implemented a prototype that automatically detects bad fixes for Java programs.

When fixing bugs, developers may have different options to design the bug fix. Leszak et al. [19] pointed out that some defects were not fixed by correcting the real error-causing component, but rather by a workaround injected at another location. An online material gives a clear description about the workaround [20]: "A workaround is a far less elegant solution to the problem. Typically, a workaround is not viewed as something that is designed to be a panacea, or cure-all, but rather as a crude solution to the immediate problem. As a temporary fix, a workaround will do very well until a suitable permanent fix can be implemented by project management personnel." Murphy-Hill et al. [21] studied why a developer might choose a workaround instead a fix at a real location. They summarized six factors: risk management, interface breakage, consistency, user behavior, cause understanding, and social factors. Some other studies also paid attention to the phenomenon of workarounds. Ko et al. [22] found that if a bug had a known workaround, developers often focused on more severe bugs. Berglund [23] indicated that bugs could be worked around and workarounds were relevant in early stages of the bug fixing process.

Different from most existing studies which investigated the design of fixes for within-project bugs, our study concentrates on the characteristics of downstream workarounds in the context of cross-project bugs.

III. Research Methodology

In this section, we first introduce how we collected data in the study. Then we present the research questions. Finally, we describe the research methods used to investigate the questions.

A. Data Source

The cross-project bugs under investigation were collected by Ma et al. [2]. The data are available online[1]. The dataset contains 271 pairs of cross-project bugs gathered from scientific Python ecosystem on GitHub. Every pair includes an upstream issue reported to the root-cause project and a downstream issue reported to the affected project. Specifically, these cross-project bugs involve 204 projects including seven core libraries in the ecosystem, that is, *IPython*[2], *NumPy*[3], *SciPy*[4], *Matplotlib*[5], *Pandas*[6], *Scikit-learn*[7], and *Astropy*[8].

Since our study focuses on the workarounds, we are only interested in the cross-project bugs for which the downstream developers have provided a workaround. In order to extract the data we needed, we manually read all the bug reports on the downstream side of the 271 pairs of bugs. If the downstream developers were willing to propose a workaround, they were very likely to leave related information in the issue reports. For example, a developer of *IPython* suffering a bug of *Setuptools* commented, "*I'll open an Issue on setuptools to deal with this, and figure out what the best workaround in IPython should be*." (ipython/ipython#8804) Two of the authors of this paper carried on this task and found 60 pairs of cross-project bugs to further investigate in this study.

For the 60 pairs of bugs, we concentrated on their downstream workarounds and the corresponding upstream fixes. Usually, the upstream issue will link to the bug-fix commits if it has been repaired. Also, if the downstream issue was worked around, the commits including the workaround would be indicated. By manually inspecting the issue reports, the two authors linked every pair of closed cross-project bugs with the commits containing the fix/workaround. Note that nine cross-project bugs have not been fixed by the upstream projects. Therefore, in total, we collected 60 downstream workarounds and 51 upstream fixes.

B. Research Questions

The aim of this study is to investigate the characteristics of downstream workarounds in the context of cross-project bugs. In particular, we attempt to answer the following three research questions:

RQ1: Are there differences between downstream workarounds and the corresponding upstream fixes?

Compared with the upstream fix, the workaround is injected in a different project and serves a different purpose. Therefore, is the design of workaround different from that of the fix? We compared them in two aspects: the code size and code structure.

RQ2: Do the cross-project bugs that downstream developers work around have some common features?

As stated, not all of the cross-project bugs have workarounds. Then what features do these 60 bugs with workarounds have in common? In RQ2, we sought to find the answer.

RQ3: Do the workarounds have some common patterns?

In RQ3, we attempted to find whether downstream developers worked around the upstream bugs in some common ways.

C. Research Methods

1) Quantitative analysis methods

In RQ1, the Wilcoxon signed-rank test and the Cliff's δ served to compare the code size between the upstream fixes and the downstream workarounds.

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used to compare whether two matched groups of data are identical [24]. The paired sample in our study are the sizes (concerning the number of modified files or the number of changed lines of code) in the downstream workarounds and upstream fixes. We set the null hypothesis H_0 and its alternative hypothesis H_1 as follows:

H_0: The number of modified files / the number of changed lines of code in the downstream workarounds is the same as that in the upstream fixes.

H_1: The number of modified files / the number of changed lines of code in the downstream workarounds is significantly different from that in the upstream fixes.

We assessed the test results at the significance level of 0.05. If the p-value obtained from the Wilcoxon signed-rank test was lower than 0.05, the sizes of workarounds and fixes were considered significantly different. Together with the

[1] https://github.com/njuap/ICSE2017
[2] http://ipython.org, https://github.com/ipython/ipython
[3] http://www.numpy.org, https://github.com/numpy/numpy
[4] http://www.scipy.org/scipylib, https://github.com/scipy/scipy
[5] http://matplotlib.org, https://github.com/matplotlib/matplotlib
[6] http://pandas.pydata.org, https://github.com/pydata/pandas
[7] http://scikit-learn.org, https://github.com/scikit-learn/scikit-learn
[8] http://www.astropy.org, https://github.com/astropy/astropy

median values of the sizes, we were able to decide whether the size of workaround was smaller than the size of its corresponding fix.

Furthermore, we used the Cliff's δ effect size to measure the magnitude of the difference between the sizes of workarounds and fixes. Cliff's δ provides a simple way of quantifying the practical difference between two groups [25]. Of all kinds of effect sizes, Cliff's δ is the most direct and simple variety of a non-parametric one [26]. By convention, the magnitude of the difference is considered either trivial ($|\delta| < 0.147$), small (0.147-0.33), moderate (0.33-0.474), or large (> 0.474) [27].

2) Qualitative analysis

For RQ2, RQ3, and part of RQ1, we performed a qualitative analysis to investigate the questions. Two authors manually inspected the issue reports and the code of fixes/workarounds for the cross-project bugs.

The two authors first individually completed the task following the same procedure and criteria. They reviewed the issue reports and code carefully, then executed the existing test cases provided by the developers to keep track of traces and to observe the input/output. During this procedure, they wrote down some necessary information: the bug information (bug type, root cause, bug impact, and participants), the bug context (related methods, test cases, traces, and input/output), and the workaround and fix strategies. And they also wrote down their findings.

After individual investigation, they came together to discuss their findings and draw conclusions.

IV. Research Results

A. RQ1:Differences Between Fixes and Workarounds

In order to compare the upstream fixes and the downstream workarounds, we first statistically compared their sizes in terms of the number of modified files and the number of modified lines of code. Then, we inspected the code structure of fixes and workarounds to see whether they were different.

Among the 60 pairs of cross-project bugs, nine of them have not been fixed in the upstream projects until now. Therefore, we could not compare their workarounds with upstream fixes. In RQ1, we only investigated the remaining 51 pairs of cross-project bugs.

1) Statistical comparision of the size

TABLE I. shows the minimum, the maximum, and the average values, as well as the 25th, 50th, and 75th percentiles of workaround/fix size. To facilitate a visual comparison, we also use boxplots to illustrate the size distributions (Fig. 1). It is clear that the number of modified files and the number of modified lines of code in workarounds are both smaller than those in fixes.

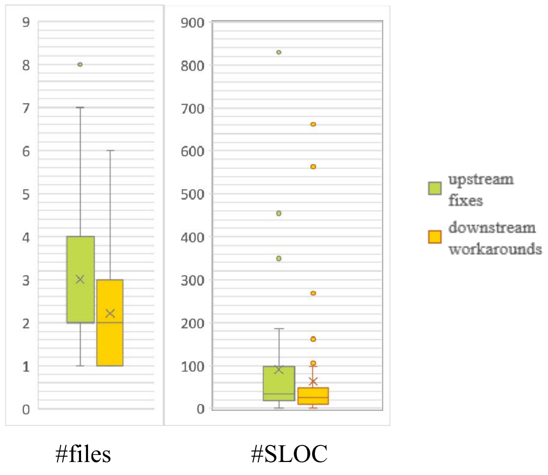


Fig. 1. Comparision of the size of fixes and workarounds

We also adopted the Wilcoxon signed-rank test and Cliff's δ effect size to statistically compare the workarounds and fixes. The results are shown in TABLE II. The p-values less than 0.05 indicate that the number of modified files and the number of modified lines of code are significantly different between the workarounds and fixes. The values of Cliff's δ mean that the difference in the number of changed files between them is small, but the difference in the number of modified lines of code is large.

TABLE II. Results of the Statistical Tests

	#Files	**#SLOC**
P-value	0.019	0.014
$\lvert\delta\rvert$	0.232	0.771

Combining the boxplots and the results of statistical tests, we conclude that the size of the workaround is significantly smaller than the size of the corresponding upstream fix.

2) Inspection of code

After statistically comparing the size of the downstream workarounds and the corresponding upstream fixes, we looked into the their code to make a further investigation.

In general, for eight out of the 51 cross-projects bugs, the upstream fix and the corresponding downstream workaround were designed in the same manner. The developers from both sides had similar idea to modify their own projects when facing the bug. For example, using the *Astropy* normalizer led to a

TABLE I. The Sizes of the Upstream Fixes and Downstream Workarounds

		Min.	**Max.**	**Avg.**	**25th**	**50th**	**75th**
#Files	*Fixes*	1	8	3	2	2	4
	Workarounds	1	6	2	1	2	3
#SLOC	*Fixes*	1	829	93	19	36	105
	Workarounds	1	662	61	10	26	45

TypeError in *Sunpy* when playing a mapcube peek animation (sunpy/sunpy#1532). It was caused by a bug in ImageNormalize class of *Astropy* which did not include a call to the inherited method autoscale_None() (astropy/astropy #4117). To address this problem, both *Sunpy* and *Astropy* used an explicit call to autoscale_None(). Fig. 2 shows the downstream workaround and upstream fix for this bug. Additionally, it is worth noting that the fix and the workaround were proposed by the same developer. Another example is shown in astropy/astropy#3052 which was caused by numpy/numpy #5251. The downstream workaround was just a copy of the upstream fix for the cross-project bug.

For the remaining 43 out of the 51 cross-project bugs, the downstream developers worked around them in a different way from what the upstream developers did to fix the bugs. This seems to accord with our intuition. Whether for within-project or cross-project bugs, a workaround is a short-term solution injected in a place other than the true root-cause location. For cross-project bugs, the workaround is placed in the downstream project where the upstream buggy method is called, while the ultimate fix is to repair the buggy method itself. Intuitively, the two kinds of modification are usually different, which is confirmed by our observations.

In Section IV.C, we will discuss the workaround patterns in detail.

B. RQ2:Common Bug Features

By manually inspecting the issues reports of the 60 cross-project bugs, we found that some bugs did have something in common. We totally identified three kinds of common features. Forty-nine investigated bugs could be classified into the remaining 11 bugs have distinct characteristics themselves and cannot be put into any category.

1) Emerging cases

A cross-project bug was reported when the downstream project encountered an emerging case that the upstream method did not cover. Thirty-nine of the 60 cross-project bugs could be classified into this kind. More specifically, we divided the 39 bugs into two subcategories.

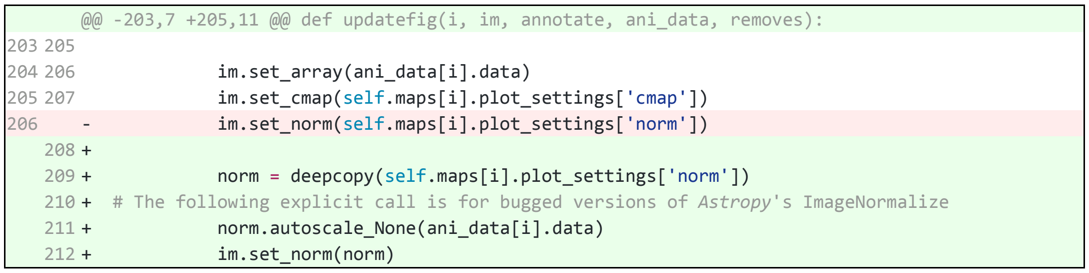
First, the original upstream method could not process certain types or forms of data. For example, astropy/astropy#3052 reported that a method in *NumPy* did not use suitable format for Unicode data (numpy/numpy#5251). Astropy/astropy#4658 was caused by np.median from *NumPy* that could not handle the masked arrays (numpy/numpy#7330). Luca-dex/pyTSA#18 worked around an upstream bug that *Pandas* could not read csv files if the column separator was not comma (pandas-dev/pandas#2733).

Second, the upstream method might not consider the processing of edge cases. For example, the method utilities.autowrap.ufuncify in *Sympy* failed when the length of the symbol list was larger than 31 (sympy/sympy#9593). The failure resulted from an error in the method frompyfunc of *NumPy*, which did not check the number of arguments (numpy/numpy#5672).

2) Wrong outputs

Sometimes, the upstream methods might produce wrong results with specific inputs which could break their downstream projects. Six of the studied upstream bugs were caused by wrong outputs.

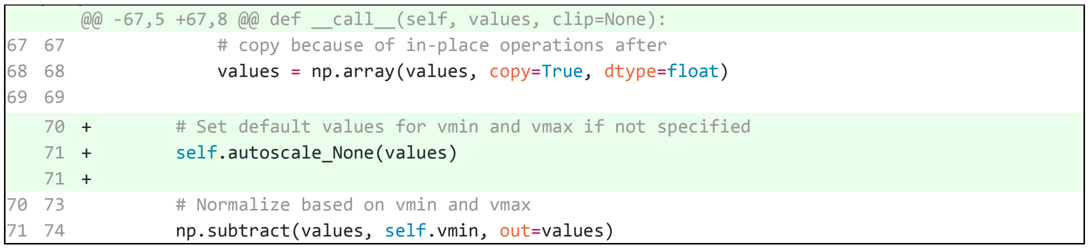
The wrong outputs are partly caused by the incorrect design of the functionality. Blaze/odo#331 was caused by the wrong output of datetime64 series in *Pandas*. The method should return NAT instead of NaN with an empty series (pandas-dev/pandas#11245). In *NumPy*, np.log1p(inf) returned NaN while it should return Inf (numpy/numpy#4225), which led to



```
@@ -203,7 +205,11 @@ def updatefig(i, im, annotate, ani_data, removes):

            im.set_array(ani_data[i].data)
            im.set_cmap(self.maps[i].plot_settings['cmap'])
-           im.set_norm(self.maps[i].plot_settings['norm'])
+
+           norm = deepcopy(self.maps[i].plot_settings['norm'])
+  # The following explicit call is for bugged versions of Astropy's ImageNormalize
+           norm.autoscale_None(ani_data[i].data)
+           im.set_norm(norm)
```

(a). The downstream workaround



```
@@ -67,5 +67,8 @@ def __call__(self, values, clip=None):
            # copy because of in-place operations after
            values = np.array(values, copy=True, dtype=float)

+       # Set default values for vmin and vmax if not specified
+       self.autoscale_None(values)
+
        # Normalize based on vmin and vmax
        np.subtract(values, self.vmin, out=values)
```

(b). The upstream fix

Fig. 2. The comparision of the code for the dowsntream workaround and the corresponding upstream fix

an undesired result in *Nengo* (nengo/nengo#260).

Some other unexpected outputs of the upstream methods were introduced by the carelessly incompatible changes when the upstream developers fixed another bug or developed a new feature. For example, the method combine_first in new version of *Pandas* performed an unwanted conversion of dates to integers (pandas-dev/pandas#3593), which made some modules of *Clair* unusable (eike-welk/clair/#43).

3) Python 3 incompatibility

Some upstream methods could not perform correctly under Python 3 while they could work perfectly under Python 2. Then, when running downstream projects in Python 3, the original upstream method resulted in a bug. For example, method loadtxt in *NumPy* failed with complex data in Python 3 (numpy/numpy#5655), which affected its downstream project *msmtools* (markovmodel/msmtools#18). Totally, four of the 60 cross-project bugs are due to Python 3 incompatibility.

C. RQ3: Workaround Patterns

After investigating the characteristics of cross-project bugs with workarounds, we summarized the common patterns from the studied workarounds. Generally, we found four workaround patterns covering the workarounds for 37 cross-project bugs.

1) Pattern 1: Using a different method

When an upstream method that the downstream project used has a bug, it is a simple way to replace the buggy one with a similar method.

Example: The *Obspy* developer experienced segmentation faults on certain systems when constructing a *NumPy* array (obspy/obspy#536). After investigation, this bug was caused by an error in np.array (numpy/numpy#3175). The downstream developers worked around the cross-project bug by using np.frombuffer instead of np.array. Fig. 3 shows the downstream workaround.

Ten out of the 60 workarounds were designed to adopt another method that could provide the same functionality. However, most of the replacements were provided by the original upstream projects. As in the example above, np.frombuffer and np.array comes from the same project *NumPy*. This phenomenon implies two things. First, some libraries may tend to develop multiple methods with overlapping capabilities. Second, the downstream projects are not willing to change their dependencies. It is reasonable since adding a new dependency means that more effort should be laid on downstream project to understand the release cycle of the new upstream project and to coordinate with it.

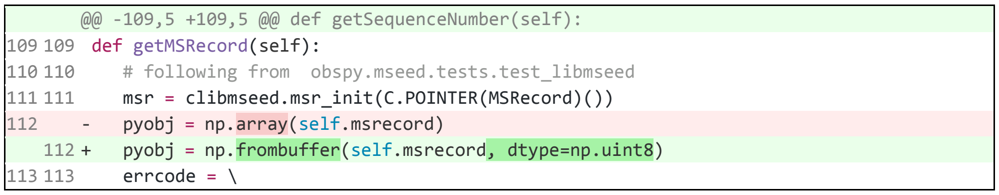
The main challenge in proposing this kind of workaround lies in two aspects. The first is to find a replacement method that is preferably designed by identical upstream project or at least a stable project. Second, the parameters should be carefully modified to fit the new method since it may require a different kind of parameter compared with the buggy method. The challenge also indicates that an automatic tool to recommend similar APIs and adapt parameters will be useful for developers to work around a cross-project bug.

2) Pattern 2: Conditionally using the original method

As we have stated in IV.B, most of the cross-project bugs are caused by one or more uncovered cases of the upstream methods. Therefore, an intuitive way to work around the bug is to only use the method in the cases that will not result in a failure.

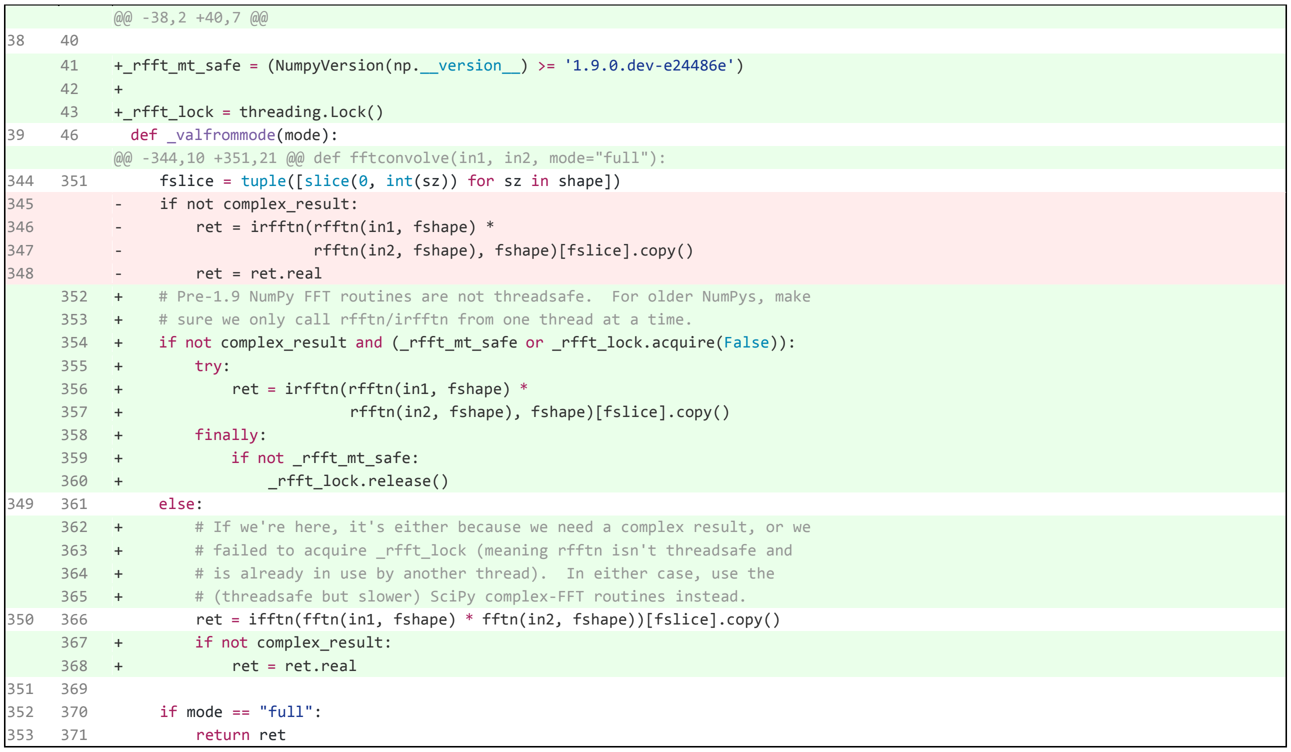
Example: Scipy/scipy#3596 recorded a bug that scipy.signal.fftconvolve did not work well in multithreaded environments. After digging into this issue, the developers found that scipy.signal.fftconvolve made use of numpy.fft.rfftn /irfftn for non-complex inputs and it was *NumPy*'s FFT routines that were actually not thread safe. Though later numpy/numpy#4655 fixed the bug in *NumPy*, the *SciPy* developers still thought that they should work around it in their side, because they support older *NumPy* version that did not have the fix. Fig. 4 shows the downstream workaround. For pre-1.9 *NumPy*, if there are non-complex inputs, *SciPy* only calls numpy.fft.rfftn /irfftn from one thread at a time to be thread safe. In other cases, they use their own FFT method instead.

However, though this workaround helped the users get out of trouble, it seemed a little complex. A developer proposed that the easiest workaround would be to convert the non-complex inputs to complex inputs (by adding 0j) so they were processed by *SciPy*'s FFT routine instead of the buggy *NumPy*'s RFFT method. This idea was disapproved by other developers. Because the *NumPy*'s RFFT method is significantly faster, it is better to use this method whenever possible. Just as another *SciPy* developer commented, "*Whatever fix is done on the SciPy side, it would be nice if it didn't prevent someone who had a new enough (fixed) NumPy from using the newer RFFT method multithreaded*."



```
       @@ -109,5 +109,5 @@ def getSequenceNumber(self):
109 109  def getMSRecord(self):
110 110      # following from  obspy.mseed.tests.test_libmseed
111 111      msr = clibmseed.msr_init(C.POINTER(MSRecord)())
112     -    pyobj = np.array(self.msrecord)
    112 +    pyobj = np.frombuffer(self.msrecord, dtype=np.uint8)
113 113      errcode = \
```

Fig. 3. The downstream workaround injected in Obspy


```
@@ -38,2 +40,7 @@

+_rfft_mt_safe = (NumpyVersion(np.__version__) >= '1.9.0.dev-e24486e')
+
+_rfft_lock = threading.Lock()
 def _valfrommode(mode):
@@ -344,10 +351,21 @@ def fftconvolve(in1, in2, mode="full"):
     fslice = tuple([slice(0, int(sz)) for sz in shape])
-    if not complex_result:
-        ret = irfftn(rfftn(in1, fshape) *
-                     rfftn(in2, fshape), fshape)[fslice].copy()
-        ret = ret.real
+    # Pre-1.9 NumPy FFT routines are not threadsafe.  For older NumPys, make
+    # sure we only call rfftn/irfftn from one thread at a time.
+    if not complex_result and (_rfft_mt_safe or _rfft_lock.acquire(False)):
+        try:
+            ret = irfftn(rfftn(in1, fshape) *
+                         rfftn(in2, fshape), fshape)[fslice].copy()
+        finally:
+            if not _rfft_mt_safe:
+                _rfft_lock.release()
     else:
+        # If we're here, it's either because we need a complex result, or we
+        # failed to acquire _rfft_lock (meaning rfftn isn't threadsafe and
+        # is already in use by another thread).  In either case, use the
+        # (threadsafe but slower) SciPy complex-FFT routines instead.
         ret = ifftn(fftn(in1, fshape) * fftn(in2, fshape))[fslice].copy()
+        if not complex_result:
+            ret = ret.real

     if mode == "full":
         return ret
```
Fig. 4. The downstream workaround injected in Scipy

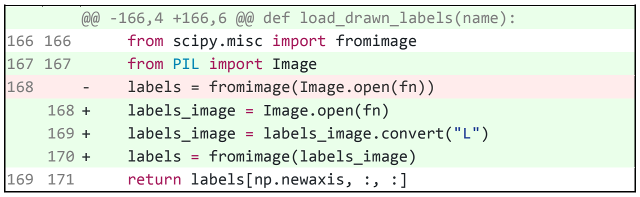
Fifteen out of the 60 workarounds were designed to restrict the use of the buggy upstream method to its covered cases. There are two key points in proposing a workaround of this kind. First, the developers should determine under what conditions the original used upstream method would fail, i.e., the uncovered cases. Usually, developers could find the answer during the process of diagnosing the bug. After that, it is important to decide how to deal with the failed cases. During inspecting the 11 workarounds, we find that the developers either made used of another method or just raised an error or an exception (e.g., sympy/sympy#9593).

3) *Pattern 3: Adapting the inputs to use original method*

To avoid the failure caused by the uncovered cases, developers may also choose to convert their inputs into a processable form which can be correctly handled by the buggy upstream method.

Example: Pyhrf/pyhrf#146 reported test failure which seemed to come from scipy.misc.fromimage. When trying to open 1-bit images, the *SciPy* method would produce a segmentation fault. In order to avoid the failure, the *Pyhrf* developers decided to first convert the 1-bit image into an 8-bit image which could be dealt with by the *SciPy* method. Fig.5 shows the downstream workaround.

Nine out of the 60 studied workarounds conform to this pattern. Though it seems to be a direct way to convert an uncovered case to a covered case in order to use the original upstream routine, this method is not always feasible.

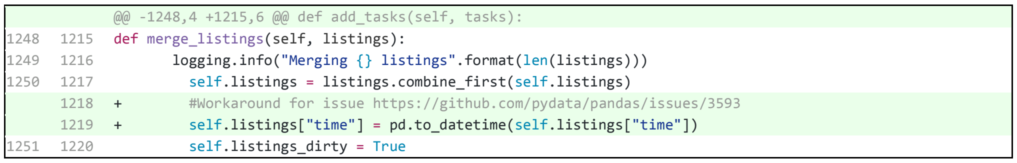

```
@@ -166,4 +166,6 @@ def load_drawn_labels(name):
     from scipy.misc import fromimage
     from PIL import Image
-    labels = fromimage(Image.open(fn))
+    labels_image = Image.open(fn)
+    labels_image = labels_image.convert("L")
+    labels = fromimage(labels_image)
     return labels[np.newaxis, :, :]
```
Fig. 5. The downstream workaround injected in Pyhrf

4) *Pattern 4: Converting the ouputs of the original method*

To work around the buggy upstream methods that produce wrong outputs with certain inputs, the downstream developers possibly choose to convert the wrong results to their desired ones.

Example: The method combine_first in *Pandas* falsely converted of dates to integers (pandas-dev/pandas#3593). To bypass the bug, its downstream project *Clair* explicitly called pd.to_datatime to convert the time-related data from integers to dates (eike-welk/clair/#43). Fig. 6 shows the downstream workaround.

Apart from this example, two other downstream projects worked around cross-project bugs in this way.



```
@@ -1248,4 +1215,6 @@ def add_tasks(self, tasks):
def merge_listings(self, listings):
       logging.info("Merging {} listings".format(len(listings)))
         self.listings = listings.combine_first(self.listings)
+        #Workaround for issue https://github.com/pydata/pandas/issues/3593
+        self.listings["time"] = pd.to_datetime(self.listings["time"])
         self.listings_dirty = True
```

Fig. 6. The downstream workaround inject in Clair

V. Discussion

In this section, we discuss the findings about downstream workarounds.

A. *Workaround Generation*

Ma et al. proposed that the workaround was the most common practice that the downstream developers used to cope with cross-project bugs [2]. Workarounds play a significant role since they can bypass the bad impact of bugs while waiting for upstream fixes, as well as shield the end user from being affected even when they use a buggy upstream version [2]. Therefore, when suffering a cross-project bugs, it will be of great use if the downstream developers could propose a workaround timely.

In Section IV, we summarized the 60 cross-project bugs with workaround into three main categories. The largest number of bugs were new cases that the upstream method could not process. To temporarily handle the problem, the downstream developers may adopt another method with similar functionality instead, limit the use of the buggy method within the cases that it can handle, or convert the emerging case to the form that the buggy method can deal with. When facing the cross-project bugs which produce wrong results with certain inputs, the downstream developers may continue use the original method, but then explicitly transform the outputs into the correct form.

Summarizing the bug types and common workaround patterns will be of help for developers to efficiently develop a suitable workaround. At the same time, it can also guide the design of (automatic) workaround generation tools. From the discussion in Section IV.C, the tool is supposed to do the following tasks. First, it can search for alternative methods which have the same functionality with the buggy method. Second, it can extract the conditions where the upstream methods do not correctly work. Third, it can adapt the input data to the suitable forms that the upstream methods are able to process.

In our opinions, a preferred workaround should follow three principles whether generated by hand or by tool. First, the workaround could suppress or bypass the upstream bug to make the downstream project run normally. Second, the workaround is supposed to make as few code changes as possible. Ma et al. indicated that the workarounds would be removed afterwards [2]. Therefore, the workaround is preferred to be designed in a way that does not affect other modules and make it easy to deprecate. Third, the workaround is supposed to use efficient methods in order not to reduce the performance of the project.

B. *Workaround Recommendation*

In a software ecosystem, some central projects are used by multiple other projects. For example, in scientific Python ecosystem, *NumPy* is the basic tool and nearly all the projects within this ecosystem depend on it. Therefore, an error in a popular project like *NumPy* may break more than one downstream projects. All of them may need to work around the cross-project bug while waiting for un upstream fix. Under this circumstance, a downstream project could benefit from another responsive sibling project which has proposed a workaround for the same bug.

Dask/dask#297 shows an example. The project *Dask* was affected by a *NumPy* bug (numpy/numpy#3484). Then a developer found that another project *Scikit-learn* was suffering the same bug. After digging into the code of *Scikit-learn*, he indicated that *Dask* could learn from *Scikit-learn*. He commented, "*Possible solution would be to add a function for python 3 compatibility, as scikit-learn did: https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/fixes.py#L8.*" Then, *Dask* copied the solution of *Scikit-learn* to their own code as their workaround for the bug.

An existing workaround in a sibling project reduces the workload of the developers suffering the same bug. However, to find a suitable workaround from another project seems to be a non-trivial task. First, the developers should find out what other projects are also affected by the cross-project bug. Then, they should get to know how these affected projects deal with the bug. Last, they have to select an appropriate workaround from these projects and adapt it to their own project. Therefore, a workaround recommendation tool which automates the process could be useful.

This tool should be designed to have at least three functionalities. First, it can predict what other projects may be influenced by the same bug and learnt the workaround from. Second, it can check for the code changes to extract downstream workarounds. Last, it can compare the context of the affected modules in different projects to rank the workarounds. The developers are facing several technical challenges to develop such a tool, which deserves a further study.

C. *Workaround Removing*

As we have stated before, the downstream workaround is a temporary solution injected in the downstream projects to cope with a cross-project bug. Unlike the corresponding upstream fix which is an ultimate and permanent solution, the workaround may be modified or discarded later [2]. We indeed find some cases which shows that the developers intend to remove or change the workarounds in the future.

Materialsinnovation/pymks#132 reported that *Pymks* broke down due to a bug in *Scikit-learn* (scikit-learn/scikit-learn#3984). The downstream developer added key word argument size as a short term solution to the current dimension requirement for the buggy method from *Scikit-learn*. He then wrote in the commit, "*Sklearn developers have already removed the dimension requirement on development version of the code. Once this version is released, this keyword argument should be removed.*" In pandas-dev/pandas#9276, the *Pandas* developer proposed a workaround for a *NumPy* bug (numpy/numpy #5562) with a comment that they would reconsider that decision once the upstream project fixed the bug. Sympy/sympy#9593 included a workaround for another *NumPy* bug (numpy/numpy#5672). The developer left a comment in the code that "*maxargs is set by numpy compile-time constant NPY_MAXARGS. If a future version of numpy modifies or removes this restriction, this variable should be changed or removed.*"

From these example, we see that the downstream developers could not decide the exact time to modify or remove the workarounds, because the time depends on when the responsible upstream projects accomplish certain tasks (e.g., releasing a new version or modify specific variables). Consequently, the downstream developers need to track the progress of their concerning upstream projects, in order to maintain their workarounds accordingly. It absolutely adds the burden of the downstream maintainers, which is confirmed by the respondents of the survey posed by Ma et al [2].

In order to reduce the maintenance burden of the downstream developers, an automatic workaround modification or removing tool is desirable. The tool is supposed to detect the occurrence of the upstream event which may influence the workaround and give a notification to the developers. Another key function of the tool is to (semi-)automatically remove the workarounds when the workarounds could be deprecated.

Additionally, the time to remove the workarounds is also worth studying. The workaround is a landmark case of the coordination between the upstream and downstream projects during the fixing process of cross-project bugs. To study the lifecycle of a workaround will help to understand how developers on both sides collaborate with each other to fix cross-project bugs and how developers from different projects cooperate within a software ecosystem.

VI. Threats to Validity

In this section, we discuss the threats to validity of our study.

The first threat concerns the accuracy of the identification of workarounds and fixes. Kim et al. pointed out that it needed high quality bug-fix information to reduce superficial conclusions, but many bug-fixes were polluted [28]. In order to identify the workarounds and fixes, two authors individually reviewed the issue reports and manually related commits indicated in the reports. They then cross-checked each other's results to maximize the accuracy of the data under investigation.

The second threat concerns the unknown effect of the deviation of the variables under statistical tests (the size of the workaround/fix) from the normal distribution. To mitigate these threats, our conclusions have been supported by proper statistical tests. We chose Wilcoxon signed-rank test and the Cliff's δ effect size, because they are nonparametric tests which do not require any assumption on the underlying data distribution.

The third threat concerns the researchers' preconceptions. The two authors that conducted the manual analysis followed the same procedure and criteria in collecting the studied dataset, identifying and comparing fixes and workarounds, as well as summarizing bug features and workaround patterns. However, it is in general difficult to completely eliminate the influence of researchers' preconceptions. In order to minimize personal bias, they discuss the results, especially the unclear cases together.

The last threat concerns the generalization of our empirical results. We conducted our study on the scientific Python ecosystem. However, cross-project bugs and downstream workarounds do not only occur within the specific ecosystem. We cannot assume that our results generalize beyond the specific environment where they were conducted. Further validation on other ecosystems is desirable.

VII. Conclusion and Future Work

In previous work, proposing a workaround is shown to be a common practice for downstream developers to bypass the impact of a cross-project bug. In this study, we studied the characteristics the downstream workarounds. First, we manually identified 60 cross-project bugs which have a workaround from 271 cross-project bugs in scientific Python ecosystem. Then, with these data, we empirically compared the workaround with its corresponding upstream fix, summarized the bug features and workaround patterns. The main findings of this study is as follows:

- In general, the size of the workaround is significantly smaller than that of the corresponding fix. The fix and the workaround usually have different code structures.
- The cross-project bugs which the downstream developers worked around are usually caused by an emerging case that the upstream method cannot process, or by a wrong output with certain inputs, or Python 3 incompatibility.
- Four patterns of workarounds are identified: using another method with similar functionality, restricting the buggy method to the range it can process, converting the inputs to a processable form, and correcting the outputs after using the buggy method.

The findings in this study also indicate the needs and possibility of developing tools supporting workaround generation, recommendation, maintenance and removal. In future work, we will continue to develop these supporting tools, as well as investigate the lifecycle of workarounds in more kinds of software ecosystems.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (61472175, 61472178, 91418202) and the National Natural Science Foundation of Jiangsu Province (BK20130014).

REFERENCES

[1] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub", *Empirical Software Engineering*, pp. 1–37, 2015.

[2] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu, "How do developers fix cross-project correlated bugs? A case study on the GitHub scientific Python ecosystem", in *Proceedings of the 39th International Conference on Software Engeneering*, 2017, p. Accepted.

[3] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub meets CRAN: an analysis of inter-repository package dependency problems", in *Procedings of International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 493–504.

[4] B. Adams, R. Kavanagh, A. E. Hassan, and D. M. German, "An empirical study of integration activities in distributions of open source software", *Empirical Software Engineering*, vol. 21, no. 3, pp. 960–1001, Jun. 2016.

[5] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: an evolutionary study", *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, Oct. 2015.

[6] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews", in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 14–24.

[7] H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects", in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 72–81.

[8] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "ELBlocker: Predicting blocking bugs with ensemble imbalance learning", *Information and Software Technology*, vol. 61, pp. 93–106, May 2015.

[9] H. Zhong and Z. Su, "An empirical study on real bug fixes", in *Proceedings of the 37th International Conference on Software Engineering*, 2015, vol. 1, pp. 913–923.

[10] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns", *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, Jun. 2009.

[11] J. Park, M. Kim, and D.-H. Bae, "An empirical study of supplementary patches in open source projects", *Empirical Software Engineering*, vol. 22, no. 1, pp. 436–473, May 2016.

[12] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast?: case study on the Linux kernel", in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 101–110.

[13] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes", *Empirical Software Engineering*, vol. 21, no. 2, pp. 605–641, Apr. 2016.

[14] J. Echeverria, F. Perez, A. Abellanas, J. I. Panach, C. Cetina, and O. Pastor, "Evaluating bug-fixing in Software Product Lines: an industrial cas study", in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 1–6.

[15] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: a generic method for automatic software repair", *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan. 2012.

[16] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each", in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 3–13.

[17] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: scalable multiline program patch synthesis via symbolic analysis", in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 691–701.

[18] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?", in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, vol. 1, p. 55.

[19] M. Leszak, D. E. Perry, and D. Stoll, "A case study in root cause defect analysis", in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 428–437.

[20] "Workaround - Project Management Knowledge". [Online]. Available: https://project-management-knowledge.com/definitions/w/workaround/. [Accessed: 08-Apr-2017].

[21] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes", in *Proceedings of 35th International Conference on Software Engineering*, 2013, pp. 332–341.

[22] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development Teams", in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 344–353.

[23] E. Berglund, "Communicating bugs: global bug knowledge distribution", *Information and Software Technology*, vol. 47, no. 11, pp. 709–719, 2005.

[24] J. D. Gibbons and D. A. Wolfe, *Nonparametric Statistical Inference*. 2003.

[25] E. a. Freeman and G. G. Moisen, "A comparison of the performance of threshold criteria for binary classification in terms of predicted prevalence and kappa", *Ecological Modelling*, vol. 217, no. 1–2, pp. 48–58, 2008.

[26] G. MacBeth, E. Razumiejczyk, and R. Ledsema, "Cliff's Delta calculator: a non-parametric effect size program for two groups of observations", *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2012.

[27] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen, and B. Xu, "Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction ? An empirical study", *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 331–357, 2015.

[28] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction", in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 481–490.