Survey Analysis

I. Ultimate Goal

The survey is designed to investigate the developers' views on cross-project bugs. In particular, we aim to:

1) find out the common practices of developers in fixing cross-project bugs;
2) inquiry about the findings that we do not understand during manual inspection of bug reports; and
3) ask for the problems and suggestions during the fixing process.

II. Survey Design

Since the upstream and downstream developers may have different concerns on the correlated bugs, we design two questionnaires for the upstream (with nine survey questions, indicated by UQ with a question number in this paper) and downstream (with ten survey questions, indicated by DQ) developers respectively. In particular, except for the last one which is an optional open-ended question, all the other questions of both questionnaires are multiple-choice questions. To further elicit the contributor's opinions, in most questions that had predefined answers, we include an optional comment box to encourage the respondents to provide the reasons of their choices or other options that we do not offer. The questions in the survey could be classified into three parts: 1) the developers' attitude towards cross-project bugs; 2) their treatment of fixing cross-project bugs; and 3) their problems and suggestions during the fixing process.

III. Target Participants

We release the questionnaires on *SurveyMonkey*. The links of the surveys[1] are sent to the participants through emails. Our selected receivers consist of two groups: 1) the developers who have ever taken part in the discussion of the collected correlated bugs (i.e., the reporters, commenters, and fixers); and 2) top contributors who have submitted at least one percent of the total commits for any of the 204 involving projects. We include the top contributors in our surveys because they care much about their projects and they are likely to pay attention to correlated bugs even if they do not participate.

IV. Respondents

In total, we successfully send invitation letters to 676 GitHub users and ask them to select a questionnaire suitable to them. Within a period of 15 days, receive overall 116 responses from the participants (32 for the upstream questionnaire and 84 for the downstream questionnaire), with a response rate of 17.2%. It is acceptable since the response rate for online surveys in software engineering is usually within the 14~20% range. Additionally, 43 participants reply the emails to show their interest in this research and provide valuable suggestions and feedbacks for our work.

V. Analysis

A. Overview

Along with the observations from the manual inspection, the data collected from the survey form the answers to the three research questions in our paper. The summary of responses for the upstream and downstream questionnaires is shown in Appendix. In general, the main source of each research question is listed in TABLE I.

TABLE I. The main source of each research question

Research question	focus	source
1	difficulty of finding the root cause	DQ2
2	important factors for tracking the root cause	DQ4
	communication	UQ3
3	practices of downstream developers in fixing bugs	DQ7
	workaround	DQ3 and DQ6
	Reporting and fixing in the upstream projects	DQ9

B. Comments

Apart from the pre-defined choices, we also provide a comment field for most questions to collect more ideas from the respondents. Since we do not require the participants to fill in the comment field for any question, the numbers of received comments for different questions vary. TABLE II and TABLE III show the statics of comments in the upstream and downstream questionnaires, respectively. The column *Question Id* indicates the specific question in the survey, the column *#comments* shows the number of comments received for a certain question, and the column *%comments* means the ratio of #comments to the total responses for a certain question. The column *type* indicates the type of the comment field. We classify our comment fields into the following four categories:

- why: to ask the respondents to explain why they make certain choices
- specific case: to ask the respondents to specify in which case they make such choices
- other: to ask the respondents to specify the concrete answers which are not provided in the options
- suggestion: to ask the respondents to give some suggestions

Totally, we receive 350 distinct comments, with an average of 11 and 28 comments for each question in the upstream and downstream questionnaires respectively.

For the upstream questionnaire, UQ4 (Compared with within-project bugs, do you pay more attention to cross-project bugs and their impact?) receives the least number of comments (four), because only four respondents choose the option "It depends on the specific bugs", the only option which needs a further comment. For the downstream questionnaire, DQ4 (What factors may act as positive roles to find the root-cause project for a cross-project bug?) and DQ7 (What do downstream developers usually do with a cross-project bug?) receive relatively less comments. In the two questions, we provide an "other" option and all the respondents who

[1] For upstream developers: https://www.surveymonkey.com/r/VSDMGSB

For downstream developers: https://www.surveymonkey.com/r/VWZX7L7

select “other” fill in the comment field for the two questions.

TABLE II. The statistics of comments in the upstream questionnaire

Question ID	type	#comments	%comments	
3	why	11	34.4%	
4	specific case	4	12.5%	
5	why	7	21.9%	Median: 11 (34.4%)
6	specific case	7	21.9%	Average: 11 (34.4%)
7	why	12	37.5%	
8	why	13	40.6%	
9	suggestion	22	68.8%	

TABLE III. The statistics of comments in the downstream questionnaire

Question ID	type	#comments	%comments	
2	specific case	41	48.8%	
3	why	47	56.0%	
4	other	8	9.5%	
5	other	30	35.7%	Median: 30 (35.7%)
6	why	35	41.7%	Average: 28 (33.3%)
7	other	14	16.7%	
8	why	38	45.2%	
9	why	24	28.6%	
10	suggestion	37	44.0%	

Though the comments only act as the complementary data for our study, we do analyze all the comments carefully. Three of the authors classify and summarize the comments of each question independently. Then they cross-check each other’s results to minimize personal bias. We value every comment from the participants because they point out the problems that the developers are facing during fixing cross-project bugs, as well as the requirements of the maintainers for their upstream/downstream projects and the issue tracking system. These comments help enrich the content of our paper (see TABLE IV) and point out the direction of our future work.

TABLE IV. The complementary data for discussion section in the paper

Focus	complementary data
Cross-project testing	UQ7 and DQ10
Notification	UQ6 and DQ8
Release management	UQ8 and DQ10
Tool support	UQ9 and DQ10

Appendix

A. The summary of responses for the upstream questionnaires.

1. Have you ever taken part in the discussion of or repaired cross-project bugs?		
Always	3	9.38%
Sometimes	25	78.13%
Never	4	12.50%

2. What proportion of bugs that you have encountered are cross-project ones?
15 18 0 67 26 20 15 7 5 24 50 15 10 5 0 14 15 3 10 25 0 10 4 5 100 10 25 10 3 25 7 10
avg: 17.28 (%)

3. As an upstream developer, do you care about the opinions from the downstream projects or communicate with the downstream developers?		
Always	19	59.38%
Sometimes	12	37.50%
Never	1	3.13%

why? (11)

Always	1. UR9: Downstream developers are consumers of our output, so they are in a position to understand when issues arise and implementation concerns when fixing them. 2. UR11: Downstream projects are major users of upstream code.

	3. UR20: Because this is why I develop libraries, it should be consumed by others and I should care about my users :) 4. UR22: Most of my software is tied together with many others. 5. UR26: It's always worth considering the feedback of your biggest consumers. 6. UR28: Downstream developers can consider the bug (or a side effect of the bug) a feature and depend upon it. 7. UR31: I do this mostly for cross-project bugs since those people have more detailed knowledge about the concerned project and its code.
Sometimes	1. UR10: So they will stop complaining. 2. UR24: We want to make sure our project is useful to others. 3. UR25: I care about downstream projects because Open Source is supposed to be easily portable.
Never	1. UR27: Only fixed one bug which I reported myself

4. Compared with within-project bugs, do you pay more attention to cross-project bugs and their impact?		
Yes	17	53.13%
No	11	34.38%
It depends on the specific bugs	4	12.50%

It depends(4)	UR11: Bugs which break previously-working downstream code are most important. UR9: Bugs usually go by severity, but downstream severity is also taken into account. UR28: fixing a cross project type bug may be more practical in a major release of an upstream project. Minor releases can be view (by some) as always working the same (bugs and all) UR24: If we break important functionality in other code, it is important to fix promptly.

5. When fixing cross-project bugs, do you take into account the downstream developers' opinions?		
Always	19	59.38%
Sometimes	11	34.38%
Never	2	6.25%

why? (7)

Always	1. UR11: They're the most intimately aware of the problem. 2. UR14: If O want my fix to be merged, I have to do it the way downstream likes. 3. UR22: I don't see any other way to be honest. 4. UR25: Fixing cross-project bugs requires information from the both sides. 5. UR31: They have more detailed knowledge about the concerned project and its code.
Sometimes	1. UR9: If I know where the discussion is taking place off-project, it's valuable information. 2. UR13: If it's not a simple fix, input from downstream developers informs how things should work.
Never	

6. When a cross-project bug has been fixed, will you notify the downstream projects which have claimed to be affected?		
Always	17	53.13%
Sometimes	10	31.25%
Never	5	15.63%

In which case you will notify them? (7)

Always	1. UR24: If we know about it, we will notify them when there's a fix.
Sometimes	1. UR2: Only if they're not aware yet or devs from those projects are not watching the relevant GitHub issue. 2. UR9: If I know where to send the notification (i.e. where the issue tracker is), then I'll usually drop a note. 3. UR10: When I remember. 4. UR13: If it's easy (e.g. cross referenced Github issue) and I'm not too busy 5. UR27: when I don't forget it
Never	1. UR25: I've never fixed one but I certainly would if I did.

7. When updating code, will the upstream project use its important downstream projects as the test beds?		
Always	3	9.38%
Sometimes	21	65.63%
Never	8	25.00%

why? (12)

Always	

Sometimes	1. UR2: because sometimes a regression is only visible by running code from the downstream project. other times it's easier to reproduce stand-alone 2. UR11: It's impossible to test against all users, but major downstream projects can be useful to try before making a release. 3. UR13: We encourage downstream projects to test when we make significant changes 4. UR16: I don't know 5. UR22: When they are light weight enough to be part of the CI sure. Otherwise the burden might not be worth it and the downstream project should test it. 6. UR24: If the fix is prompted by a particular failure downstream, checking with downstream is the easiest way to verify that it is actually fixed, and nothing else is broken. 7. UR25: It's good to know whether or not the bug impacts downstream too. 8. UR28: It is often impossible to know the complete list of downstream projects. And then each project has a different way to run its tests. Standard ways of running tests would help. Discovering complete lists of downstream projects will probably always be impossible.
Never	1. UR3: Too much overhead for testing. It's better to write a small unit test. 2. UR9: We can't cross-test against every library that consumes our data, it would be too difficult and time consuming. 3. UR10: It is up to them to test betas. 4. UR21: We reimplement a minimal unit test exposing the bug and then add it to the test suite.

8. When scheduling a bug-fix release, will you consider the requirements of the important downstream projects?		
Always	8	25.00%
Sometimes	22	68.75%
Never	2	6.25%

Why? (13)

Always	1. UR11: Important breakages downstream can expedite bugfix releases. 2. UR26: Usually keeping downstream projects working means keeping the same API. In that sense It's important to consider them.
Sometimes	1. UR2: there's quite some effort involved in making a release, so we only make a bugfix release quicker if the downstream project really needs the fix 2. UR9: On the margin when deciding whether to release immediately or give a bit of time for reflection, this argues in favor of a quick release. 3. UR13: If downstream developers make a compelling case that a fix is important. 4. UR16: I don't know 5. UR22: Same as above. 6. UR30: Typically not. We make releases when we are ready. 7. UR25: Bug fix in upstream impacts downstream too. It really depends. 8. UR28: It is a numbers game. If the problem is really a bug and needs to be fixed but will break almost everything then hold off until a major release. Otherwise how many downstream projects break when the bug is fixed? Is the bug now a feature and will never be fixed? Most of this decision process lies with the developers of the upstream project. There are good reasons to fix all bugs and good reasons not to. It depends. No bug fix release can ever claim to not break anything (unless it makes no changes at all). 9. UR31: This would happen rarely if I feel the downstream project has very legitimate reasons. 10. UR32: The reformation should help best of both-ends.
Never	1. UR14: I freeze the environment.

9. What suggestion do you have for fixing cross-project bugs effectively and efficiently (in terms of GitHub issue tracking system, maintenance activities, developers' practices, or anything that you think will be useful) (22)

1. UR19: Downstream projects should have an automatic continuous integration run of their tests using the latest development version of all upstream projects they depend on.
2. UR16: Get support from github to e.g. mirror the issue, run downstream test suite etc.
3. UR14: Add cross-project integration tests as separate Travis jobs / projects
4. UR13: Knowing developers on other projects in person, e.g. by meeting at conferences, makes it easier to discuss potential cross-project bugs, and work out answers without getting hung up on whose 'responsibility' it is
5. UR12: Open issues among the various affected projects and add GitHub reference links to the related issues in the source/downstream projects.
6. UR11: GitHub's issue cross-linking is extremely helpful in this regard. Downstream project issues can link directly to the upstream issue, preventing the need to rehash all the details.
7. UR10: Need more cross project developers.
8. UR9: Tracking down upstream bugs and providing bi-directional links (mostly happens automatically with Github) is very useful.
9. UR6: Improved downstream/upstream bug referencing visualization or summaries on the Github bug page (to improve the current inline "this bug has been referenced"). This would require it to be possible for projects to mark other projects as "upstream" of them on Github (either manually, or automatically from imports).
10. UR5: Release bugfix often so people can use them
11. UR3: None. Gitub issues and mailing lists have worked well.
12. UR2: There's not much focus on or tools for automated multi-project testing. This is the only good example I can think of: https://github.com/pv/testrig
13. UR32: For the given seven topical tools, open a separate github issue tracking system which can be traced down from respective tools issue listings.
14. UR31: Developers from both the project should participate in the discussion on the issue tracker.
15. UR29: Perhaps a periodic Travis CI run encompassing downstream would be useful.
16. UR28: Here is one specific example. I don't have any idea how github would/could implement it. But it is a real world example of a useful collaboration that is not currently possible with github. The project swig creates a tool that is a compiler of interfaces between C/C++ and many (a dozen or more) scripting languages. This tool (swig) is used by countless other projects to create scripting language interfaces. It has lots of issues with cross-project

bugs. Bugs that some consider bugs and others consider features (or the side effects of the bugs to be features). Recently a seemingly harmless change was done to one area of swig to fix a bug. Swig itself has a rather large test suite (that is run on/through github). It is large (takes hours to run) and it all passed. Yet a non-trivial amount of downstream projects test cases did not pass. They depended upon the bug still working as a bug. The only way for a swig developer to know about these problems was trough the issue tracker. An issue tracker is one step better than email. And does not help a lot in knowing that a cross-project bug even has the potential to exist. Redhat to the rescue! A very generous redhat developer had setup a build tree that would build and run the tests suites of every single project that depended on swig (via rpm dependencies). The developer then gave feedback (sometimes specific erros) on which projects built clean and which failed with the modifications to swig. In some cases the swig developers were able to send patches to the downstream developers (through redhat). In others swig was modified (makeing the bug a feature because too many projects depended on broken behavior). And in some cases the downstream project was advised that what they were doing was so far out of the box that they would have to figure out another way to do things. The big point in all this is that not just the upstream tests were available to the swig developers. Many many more downstream tests were now visible. So, the impact of changes could be known. If git hub could come up with some sane way of running these combined builds/tests it would be awesome. The big "mega-test" could be run say by a project administrator before a potential public release. If the mega test were run for each commit there would be blood in the streets. And of course there would need to be some sane way to link the projects that are being tested together. Something along the lines of how a linux distribution picks a list of packages to be put into a group. Simply linking the master branch of each project to all be tested as one would also be wrong. Maybe an idea like this is only practical for a creator of a linux distribution. But it would be cool if github could link project tests in some similar way.

17. UR26: When pinning a project to an upstream project, it's important to consider how an upstream maintainer handles bug reports and fixes, as well as handling a stable API.
18. UR27: Dependencies which GitHub currently does not support
19. UR25: For fixing cross-project bugs, downstream and upstream should co-operate when possible.
20. UR24: When projects run tests with Continuous Integration, install prereleases of dependencies. With pip, this is: pip install --pre This will allow upstream projects to prepare a prerelease, and give downstream projects a chance to notice breakage in their CI before a final release in the upstream package is made. Sometimes this is a downstream fix, others an upstream fix, but it is good to notice before the release is final.
21. UR22: Not sure. Maybe when projects are really close a relationship will be developed naturally. GitHub could add a "dependencies favorite" on each project to easy the web navigation. That would be a start to improve communication between projects.
22. UR20: Crosslinking bugs and issues is a key here. Also voting on bugs and collection opinions is very important.

B. The summary of responses for the downstream questionnaires.

1. Have you ever taken part in the discussion of or repaired cross-project bugs?		
Always	3	3.57%
Sometimes	72	85.71%
Never	9	10.71%

2. Is it difficult to find the root-cause project for a cross-project bug?		
very easy	1	1.19%
easy	5	6.17%
normal	27	32.14%
difficult	46	54.32%
very difficult	5	6.17%

Usually, how long does it take to find the root-cause project for a cross-project bug? (41)

very easy	1. DR12: 1 hour
easy	1. DR1: 1 hour 2. DR5: A few hours 3. DR30: 1-5 hours 4. DR39: couple of hours 5. DR82: minutes
normal	1. DR9: 30 min 2. DR24: variable 3. DR26: 30 min 4. DR34: 2 hours 5. DR35: 1 hour 6. DR36: 3 hours 7. DR42: Varies 8. DR48: 2 hrs 9. DR56: 1 day 10. DR59: Almost less than a day 11. DR61: 45 min 12. DR68: Depends on the bug 13. DR69: same as debugging the project itself 14. DR75: very variable, depends on lots of factors!
difficult	1. DR3: Few days to weeks 2. DR6: 1 week

	3. DR7: 1 day 4. DR10: it can take an hour or two 5. DR11: 1 hour 6. DR13: hours or days 7. DR19: a couple days 8. DR26 ~4 hours 9. DR27: A day or two 10. DR32: 2-3 days 11. DR37: At least a day or two 12. DR43: About one week. 13. DR44: Depends on the bug. 14. DR49: 1 month 15. DR54: 2 hours 16. DR60: about 4 hours 17. DR71: 2 hours 18. DR72: weeks/months to get agreement 19. DR73: Usually more than an hour. 20. DR80: A day or two 21. DR53: hours
very difficult	

3. Compared with within-project bugs, do cross-project bugs have more severe impact on the downstream project?		
Yes	34	40.74%
No	8	9.26%
It depends on the specific bugs	42	50.00%

Why? (47)

Yes	1. DR3: It is hard to find out what is going wrong. 2. DR6: Complexity, usually bugs result from interoperation/different assumptions. 3. DR7: Since the bug "carries over" it becomes difficult to trace the bug back to its root through recursion. 4. DR9: takes more time to fix 5. DR19: One has to make sure that the bug really comes from some other library and not from your own project. 6. DR20: Cannot rely on a fix in the next release as lots of users still might have an old version of the upstream lib installed. 7. DR22: Most numpy bugs we have had to deal with only impacted rare astropy use cases. 8. DR24: The ripple of identifying, building a workaround while waiting for the bugfix to propagate, and then undoing the workaround is an inevitable time sink 9. DR25: You usually consider the upstream project to be "perfect" 10. DR27: Waiting for an upstream fix delays the release of a downstream fix 11. DR28: Fixes have to be implemented in the upstream project, which creates a dependency on those project maintainers and their update. 12. DR31: Cannot fix ourselves and this prevent a bugfix release 13. DR36: Have to ask maintainer for fix in another repo 14. DR38: Have to wait for an upstream release for the bug to be fixed 15. DR39: It's hard to find it out. 16. DR59: You can never tell at how many different place the downstream project gets damaged by the bug. You can't always guard against all of them. 17. DR65: We need to wait for a release cycle to get the fix - sometimes a major release cycle. We also usually need to work around the bug on older versions, which means adding version-dependent code to the project, increasing maintenance burden. 18. DR68: Differences in release cycles, the presence of the bug is dependent on the version of the dependency 19. DR78: Changes upstream can have subtle unforeseen consequences 20. DR80: If not involved with the upstream project, this involves waiting for the dependency to be updated, taking more time. 21. DR83: You have to coordinate workarounds in your own code with releases of another project. 22. DR53: Because workarounds for the underlying code must be made, and preferably the workarounds are only used in the situation when the bug still exists in the upstream project. For within-project bugs this is not necessary since all parts can be updated simultaneously. 23. DR52: They can't be fixed without involving the upstream team. Often a workaround must be implemented if the upstream team is not willing or able to fix the bug quickly. 24. DR82: It does depend on the responsiveness of the upstream project. I find myself writing fixes within the downstream project more often then fixing upstream and needing to work with an external team.
No	1. DR11: Because in a world of open source software and good package managers the problem can be detected from the correct dependencies and usually the fix is quite simple because most of the cases it's just a new case not covered 2. DR61: Most of the time a downgrade in the upstream project can be used as a workaround 3. DR67: Fixing the bug is harder not more impactful.
It depends	1. DR8: At a minimum you need to implement a workaround until the upstream bug is fixed. 2. DR12: Sometimes an external fix is doable, sometimes not. 3. DR13: The bugs are often things like, "this function has an annoying corner case" 4. DR18: sometimes there is an easy work-around

	5. DR30: They rarely lead to wrong results, which is the most severe kind of bug. They can often be worked around easily. 6. DR32: Not easily fixable, sometime require backport/hack 7. DR34: For a bug in numpy, that's used throughout my project so the impacts are severe. For a bug in matplotlib or scipy, that might only affect a small part of my project. 8. DR35: some can be fixed by changing options 9. DR37: Most are easy to work around, though the resulting code is not always very attractive. 10. DR41: because. Sometimes it is harder to fix because other downstream projects depend on the behaviour; sometimes it's just a small workaround. 11. DR42: Not all bugs are the same. 12. DR43: Usually the bug is subtle and affects some very specific function. It may be difficult to bypass but anyway, the impact can be confined in a small scope. 13. DR44: A bug impacting performance is not the same as a bug impacting functionality. 14. DR48: I think it depends on the level of dependency on that upstream project. For example scikit-learn having a major dependency on numpy can lead to more severe bugs if there is some bug in numpy. 15. DR50: not all cross-project bugs have severe implications 16. DR60: it really depends on the bug 17. DR66: If an upstream bug affects my project, I have to wait until the upstream patch is released before my project is fixed. 18. DR72: how central the impacted functionality is to the downstream project 19. DR73: Cross-project bugs aren't more severe than in-project bugs. They're just harder to find. In terms of the overall impact on the project, cross-projects bugs are much less important than within-project bugs. 20. DR75: the impact is the impact! it depends on the case

4. What factors may act as positive roles to find the root-cause project for a cross-project bug?		
The traceback that the reporter provides	62	73.81%
The communication with the developers of the suspected upstream project	61	72.62%
The familiarity with the downstream project and its dependent ones	53	63.10%
Other (please specify)	8	9.52%

Other (8)

1. DR3: Systematic testing usually allows one to narrow down the problem to an upstream bug. 2. DR9: documentation of the upstream project and comments in code 3. DR24: One is rarely facile with the upstream project's internals, so communication is essential 4. DR30: Familiarity of the downstream package with the upstream one 5. DR49: stackoverflow 6. DR66: live debuggers, viewing the source code 7. DR73: A clear distinction between errors caused by upstream and downstream code. 8. DR75: having a simple&reliable test case is a great help, hopefully the submitter will have done this, but I might have to

5. Some cross-project bugs are reported to the downstream projects even when their root-cause projects are identified. Then why do reporters still submit them to the affected downstream projects? (e.g., obspy/obspy#536)		
To require a local workaround in the downstream project in order to avoid the bad impact	60	71.43%
To simply record the bug information as part of the project history	34	40.48%
To track the fixing process in order to help those who later encounter the problem to find the patch	54	64.29%
Other (please specify)	30	35.71%

Other: (30)

1. DR9: reporters prob don't know it's an upstream issue 2. DR10: When the bug is first entered on the downstream project's issue queue, the author of the bug report may not know that the downstream bug is caused by an upstream bug. 3. DR12: Users do not necessarily know the root of the problem and simply report it where they observe it. 4. DR14: To find a work-around. 5. DR15: Usually bugs are found by users who do not necessarily know the cause of the bug. 6. DR17: Because that is the project they are using and they are not aware of the upstream. 7. DR24: they are not aware of the dependency 8. DR26: Because they don't really know that it's an upstream bug they just know the tool they are trying to use isn't working 9. DR27: End-users often cannot locate the source of a bug, only note its end effects 10. DR30: More familiarity with the downstream package 11. DR35: because it is encountered there 12. DR40: You need deep expertise to identify where the true source of a bug is 13. DR41: Because they only observe the problem (which is in the downstream project), not the cause. 14. DR42: Depends entirely on the bug. Sometimes upstream won’t fix, sometimes upstream won't acknowledge, etc. 15. DR45: They could also not be aware that the bug is upstream 16. DR55: Reporter being unaware of upstream fix 17. DR57: ignorance

18. DR58: Sometimes people simply don't know better. It's not always that one can find out where the bug came from.
19. DR59: it's easier to create an issue than it is to determine its cause, so people are sometimes unaware
20. DR65: They may not be aware that the issue they are seeing is actually in an upstream project.
21. DR67: because that is the tool / library they are interacting with
22. DR68: Many users don't realize where bugs come from, and report to the project that they encountered it in.
23. DR69: Because bug reporter is unfamiliar with upstream bug reporting procedures
24. DR72: end users don't know about dependencies
25. DR74: Lazy or just don't know where they should report it...
26. DR75: because it's difficult to know what's appropriate initially
27. DR78: The reporter may be unfamiliar with the culture of the upstream project and hope that downstream maintainers will take care of the communication. In a sense, it is ultimately the responsibility of the downstream project, since that has chosen to adopt the upstream and indeed could choose to drop it.
28. DR81: Don't recognise the real source of the bug
29. DR84: Because the bug reporters are unaware that this is a cross-project bug.
30. DR82: Because the downstream project is the immediate concern. I think most cross-project bugs are pushed up the stream of dependencies not directly tackled at the root.

6. Compared with within-project bugs, is it more difficult to deal with cross-project bugs in order to eliminate the ill effects on the downstream projects?		
Yes	53	63.10%
No	3	3.57%
It depends on the specific bugs	28	33.33%

Why? (35)

Yes	1. DR1: workarouds dependent on the versions 2. DR3: It takes time (months) before a version of the upstream project with bugfix is released. 3. DR5: Because you usually have to code a work-around for buggy and non-buggy versions of the upstream code. 4. DR6: Working with effects on a less-well known system. 5. DR15: It is out of your control. 6. DR20: Familiarity with code 7. DR24: At the minimum, there is an n-fold penalty to be paid retesting the downstream projects, with potentially more work eliminating workarounds. 8. DR26: Lack of familiarity with the upstream project. Lack of control of the upstream project. 9. DR37: Within project we control releases, but with dependencies a user might have an old version installed, so we have to work around bugs regardless of if they're already fixed upstream. 10. DR38: Need to wait for upstream release or provide an awkward work-around 11. DR39: Because you have to wait until someone will fix bug in upstream, or merge your PR with fix to upstream. 12. DR43: It is especially annoying when the development pace differs much between the upstream and downstream projects. Downstream projects usually iterate rapidly, while the bug-fixing process in the upstream may be sluggish 13. DR48: Its more difficult to find them. 14. DR50: fixing the bug may affect other downstream projects 15. DR60: it requires you to update two projects, with at least a version bump in the downstream project 16. DR65: The release cycles of downstream and upstream projects are out of sync 17. DR67: you can have multiple downstreams and you require communicating with a larger amount of people to fix these bugs 18. DR68: As above, difference in release schedules. We need a workaround for old versions of the dependency, but can use the new dependency as is once fixed. 19. DR53: Because there may be subtle but important changes caused by fixes in the bug in the upstream project so backwards-compatible solution may be needed (or temporary deprecation warnings) 20. DR52: Yes, see above - fixing the bug involves two project teams. 21. DR82: You need to be cognizant of not breaking downstream functionality for others
No	1. DR78: Unit testing. Unless downstream rely on unspecified behavior, but then that's their problem
It depends	1. 2.8: Even if the upstream bug is fixed, we can not rely on the fix (compatibility with old versions) so we need to implement local fixes. 2. 2.9: Before submitting bug to the upstream project, we have to make sure this is not a downstream project bug and we need to narrow down the bug to a minimal script 3. 2.12: If it's hard to work around, then I have to wait on upstream's release schedule for a fix to appear. 4. 2.19: same as above... 5. 2.23: Mainly becaus of the inertia of the upstream project (long tests, heavy procedures, etc) 6. 2.24: what is this question even asking? 7. 2.25: The difficulty depends on the readability of the upstream source and on responsiveness of its maintainers. 8. DR57: Some bugs have easy workarounds, some upstream bugs are fixed very quickly in minor releases, but some bugs are difficult to handle 9. DR66: Some project maintainers are more open to patches than others. Some projects have different release cycles which can slow the resolution of a bug. 10. DR72: need to convince upstream project that bug is important to fix, may not be to their priorities 11. DR73: It depends on how easy it is to make a workaround, and/or how responsive upstream developers are to fixing the issue. 12. DR75: not sure, haven't worked on enough to say

	13. DR76: Small cross-project bugs could get small workarounds in the scope of a downstream project

7. What do downstream developers usually do with a cross-project bug?		
Doing nothing but waiting for the upstream developers to fix the corresponding upstream bug	28	33.33%
Restricting the dependent upstream versions to those without the corresponding upstream bug	42	50.00%
Proposing a temporary workaround locally to bypass the problem	75	89.29%
Using a different upstream project which provides the same functionality instead	7	8.33%
Other (please specify)	14	16.67%

Other: (14)

1. DR17: Help fix the upstream bugs
2. DR18: work around the upstream bug by using un-affected functionality
3. DR22: Actively help the upstream project by proposing/pushing solutions
4. DR28: Report to upstream project and help to fix.
5. DR39: Proposing a fix to upstream
6. DR41: Fix the upstream bug :-) // curse the upstream devs... :-)
7. DR42: Depends on the bug
8. DR43: report and help the upstream developers fix the bug
9. DR45: Sometimes they are also part of the upstream projects, and try to fix the bug
10. DR48: Report bug to the upstream developers and if feasible help them with the fix.
11. DR59: Sometimes they send a patch to the upstream project and fix a bug. If the patch isn't accepted for a long time, the developer may also want to maintain a fork of the upstream project which contains the fix. This adds to a lot of cruft within the ecosystem. There's no easy solution to this, since even upstream maintainers may have good reasons for not accepting a particular fix.
12. DR67: Fix the bug upstream
13. DR73: Communicate with upstream project developers to help/encourage them fix the bug.
14. DR75: whatever is easiest in their specific circumstances, above are good examples! but probably work around the issue

8. Is it necessary for the affected downstream projects to be notified when the upstream developers fix the corresponding upstream bug?		
Yes	58	69.05%
No	26	30.95%

Why? (38)

Yes	1. DR22: So that work-arounds can be used only for affected versions of upstream 2. DR24: Duh 3. DR28: Mainly if it was an issue reported by the downstream project 4. DR30: So downstream knows in which version the bug is fixed upstream, and so know how long we have to maintain our own backport of the fix. 5. DR31: Remove the workaround 6. DR33: To be able to properly communicate to the downstream's community; also to remove, as needed, any workaround. 7. DR39: Downstream developer have to know when to update his code to keep it as clean as possible, without redundant workarounds. 8. DR42: Coordination 9. DR45: Many reasons. Downstream developers could clean the workarounds for their code 10. DR57: relevant to know whether bug is temporary or requires permanent changes 11. DR60: it's helpful 12. DR62: So that pending issues can be closed and users can be notified of the fixes in subsequent releases. 13. DR67: remove workarounds or other mitigation 14. DR68: Probably don't need to be notified directly, usually the downstream project will watch the issue for changes (github makes this easy) 15. DR72: helpful more than necessary, downstream project members are likely to be tracking the bug in upstream project 16. DR53: To make sure any workarounds are still valid 17. DR82: So they can manage their project well
No	1. DR3: Just to read the release notes if you care. 2. DR7: Numpy has millions of downstream projects. The downstream maintainers should keep track of their bug with numpy just releasing the fix in the changelog 3. DR8: Downstream devs can follow the upstream issue on their own. 4. DR9: not explicitly -- as long as it's part of a version changelog it's fine just to note it there 5. DR12: Workaround already in place. 6. DR13: The downstream often cares a lot. 7. DR25: too many projects depend on numpy for example, it just imposible 8. DR27: automatic libraries.io notifications handle this pretty well 9. DR37: Typically we read upstream changelogs for big changes; for smaller things, we'll either make or track the issue and will be notified when it's fixed and closed. 10. DR41: It's nice, but usually I'm subscribed to the upstream bug 11. DR46: necessary in what sense? no, it's not the law. 12. DR47: The fix should be documented in release notes of the later version and at the bug-tracker comments within the project. On GitHub

	the downstream developers can watch and participate in issue discussions and get notified when the bug gets resolved. 13. DR50: not feasible for projects with many downstream users 14. DR56: Publicly announcing should be sufficient, downstream developers should keep track of dependencies announcements. 15. DR59: I wouldn't say it's necessary. It would be good to have. Normally the downstream project developers just notified on the given issue page, but this doesn't strictly have to happen. 16. DR66: It does help, but is not necessary. It's infeasible to notify all downstream projects regardless of size. 17. DR73: Not necessary, but very helpful, so that any workarounds can be removed. 18. DR75: it can be helpful and might happen automatically when the same people are involved in both 19. DR78: It behooves those depending on a project to keep abreast of its developments; perhaps via their own regression tests. 20. DR80: I don't believe it's the responsibility of upstream projects to notify downstream projects of a bug resolution. It would be nice, but should not be expected. 21. DR83: It's helpful but not strictly necessary.

9. Have you ever helped or are you willing to help the upstream developers to repair the upstream bugs which have endangered your projects?		
Always	30	35.71%
Sometimes	52	61.90%
Never	2	2.49%

why? (24)

Always	1. DR6: Open source, collective benefit 2. DR34: Self-interest--it makes my project work better. 3. DR59: They don't really owe me anything. If I'm affected, I should be willing to put in effort too. 4. DR66: Fixing a bug that the upstream developers don't know about helps everyone. 5. DR53: Hope that the solution gets fixed sooner than later.
Sometimes	1. DR22: It has depended mostly on how easy it was to understand why the bug happened 2. DR24: Time constraints. This may be improving with modern development cycles. 3. DR27: It depends on how much expertise is required to fix the upstream bug. 4. DR37: It's difficult to jump into a project (even one you use a lot) and learn their code, PR process, testing systems, etc. So there's a big learning curve, but it feels great to contribute to a big project you use a lot. 5. DR39: If I think I can find a bug in upstream code, and fix it - I always try to do so. By this I make upstream project better, and I also like when my Name is in contributors list of some serious project :) 6. DR41: Because it's open source. It's fun to battle the hard bugs and get to know new code bases. 7. DR48: Depends on what technology they are using... if I'm working on a downstream project which is meant to act as a wrapper, I might not want to contribute to the project. 8. DR50: depends on how much I know about the upstream project's internals 9. DR51: in most of the cases downstream developers find the bug easy to understand and fix 10. DR60: a sense of obligation 11. DR62: I don't have the necessary skillset. 12. DR65: If the fix is simple I'm happy to propose it. Diving into a new codebase can be a very costly endeavor so if I need to understand a huge part of a new codebase I'd rather someone with more experience fixed the issue. 13. DR69: I usually lack the experience with the upstead code. 14. DR72: sometimes don't have enough knowledge of upstream project to be able to help. Have often at least done testing & reproducible example of bug. 15. DR75: generally, have forked several repos to do this. depends on what else is going on and how current/well maintained everything looks 16. DR78: Sometimes it's too hard, I might not have the expertise. 17. DR80: If it something in my expertise or capability, I have/will help in fixing the bug. 18. DR52: I do so when I understand the upstream code sufficiently well.
Never	1. DR5: Often out of my expertise.

10. What suggestion do you have for fixing cross-project bugs effectively and efficiently (in terms of GitHub issue tracking system, maintenance activities, developers' practices, or anything that you think will be useful) (37)
1. DR20: Ability to link issues cross-project and only mark the issues as resolved if all linked issues are resolved 2. DR15: In general communication is very important. 3. DR12: Nothing specific: writing the usual bug report in the project to which the bug belongs. 4. DR9: github issues, with their cross-project notifications, actually work really well. it seems to me it's just a personnel problem (limited time for contributors, who are often volunteers doing this on the side, to fix a bug) 5. DR7: GitHub should have an issue referencing system across projects so that when the issue upstream is fixed, a notification goes to the downstream project automatically. Same the other way round while reporting. 6. DR5: Good software development practices and personal relationships between devs. 7. DR6: Improved downstream/upstream bug referencing visualisation or summaries on the Github bug page (to improve the current inline "this bug has been referenced"). This would require it to be possible for projects to mark other projects as "upstream" of them on Github (either manually, or automatically from imports).

8. DR3: The main difficulty is finding someone to fix the bug in an upstream project in case I'm not familiar with the upstream source code.
9. DR2: When the downstream and upstream projects are both on Github, things are quite easy to track. When a project exists elsewhere, some manual tracking is required. I worry that this creates an almost unbeatable network effect around Github -- we need tools to integrate with other bugtrackers so that we make sure we have choices in the future.
10. DR1: release often
11. DR51: downstream projects should always specify the compatible upstream dependencies and upgrading this specification should also go through thorough examination as a pull request. downstream developers may come forward and contribute to upstream projects fixing cross project bugs.
12. DR50: Prompt and clear communication between project developers
13. DR48: Its more important for the upstream developers to check on their downstream clients for any bugs... and they do that, if any bug is found to be like that, it should be tagged as "cross project bug" and the depending library should be notified.
14. DR46: The ability to link to bugs in related projects is very helpful. It would be nice if the testing suites for downstream projects were run before releasing upstream version that contain breaking bugs. For example, if numpy developers were required to run the testing suites of scikit-learn, etc. many cross-project bugs would be effectively avoided before release.
15. DR44: A dependency system for bugs would be nice.
16. DR42: A better way to link cross-project context: (issues, etc.)
17. DR41: It would be nice to have issue dependencies, so I don't have to subscribe to upstream bugs but get a notification if a bug is fixed and I can now act on it (irrelevant of inter or intra project issues)
18. DR38: Getting notification for the release containing the fix to an issue, or a given pull request, would be useful.
19. DR37: Github's made this process really nice already, in that for most projects you can raise an issue and have a discussion with the developers, which either fixes the issue, or gives you the tools to fix the issue yourself. Before Github, it was significantly more difficult, I think, so moving projects to Github or similar locations makes a big difference.
20. DR34: Use labels in the GitHub issue tracker to mark issues that are caused upstream.
21. DR32: Readable traceback; Better tracking of downstream projet: link the upstream issues from the downstreams project to report interest in a quick fix; Maybe some cross project CI for interconnected projects
22. DR31: Automatic notification for downstream project that have subscribed to a bugfix issue
23. DR30: Seems to work reasonably well. The main issue is not being able to bump dependency versions easily, and I don't see a way to work around that.
24. DR28: Cross-project issues would be nice to include all relevant people in the discussion easily
25. DR78: Clean interfaces on both sides, good documentation, unit tests, ... Those are all pretty obvious, I suppose, and not directly related to cross-project bugs but I believe they would help preventatively.
26. DR75: having a friendly response from the maintainer(s) helps a lot with getting started
27. DR72: reference upstream bug report in downstream issue tracker. Realise that the bug might not be high priority for upstream project. Be willing to take time to create reproducible failure and to test fixes provided by upstream project.
28. DR71: Ability to migrate issues from one project to another.
29. DR69: In github: Post a message in a thread if the status of a mentioned other issue (within-project or cross-project) is set the "closed"
30. DR63: GitHub issues can be linked cross-project and tracked together
31. DR62: Maybe an issue tracker that links across projects, a system that records the findings of root cause analysis done by the project maintainers/developers. I think the most important thing is collaboration between developers of the upstream and downstream projects.
32. DR61: Upstream projects have to make it very clear which bugs were fixed in a release of their project.
33. DR59: Nothing specific. GitHub issues, IMO, work really well.
34. DR58: smarter inter-project "blocked by" status
35. DR57: Maintain communication, provide early feedback and test cases to fix bugs. (More importantly. continuous integration tools and testing of pre-release versions avoid many bugs in the first place.)
36. DR52: Have cross-references in the issue trackers on both sides, this helps everyone understand what's going on and who suffers from which problems.
37. DR82: Possibly being able to link bug reports in one project to another. Or have a single bug report/comment thread span multiple projects