| D# | Answer | Guidelines |
|---|---|---|
| **D1** | a. When **clarifying the main thought and flow of the code logic [C1]**<br><br>b. When **there are bugs existing [I5]** | *C1. If it provides context to the reader<br>*I5. If it has high priority |
| **D2** | I don't think I have definitive guides on choosing one over another, though, so it's mostly a "gut feeling". **I would say that I would use a FIXME-like comment for something local to the code where the comment is (like a rare edge case not handled which should be handled near the comment) [C3]**, while **an issue or tracker is for some bigger beast, like "refactor this class" or "change the way those classes interact" [I3][I6]**.<br><br>Now to your examples.<br>Example #1 is not exactly a "to do" comment, but an explanation why do we need such a weird code after the comment. I don't think there's better way of explaining a "this is weird because we workaround that bug" thing, plus it fits perfectly to **"fix the stuff in the near"** criteria I've written above.<br><br>Example #2 is something I would not call a TD (technically speaking), as in my point of view a TD is something being deliberately done wrong or inefficient to cut development costs in the short run. In case of #2 it wasn't exactly so – I needed a variable that I can change during runtime and subscribe to its changes in the code, so It was more like a "feature tracker". But even if we would classify it as a TD (which it partially is, I agree) you see that **fixing it spanned multiple files and involved touching quite a lot of places in the codebase [I3]**. | *C3. If it has a local scope<br>*I3. If it spans to multiple places<br>*I6. If it requires medium/large effort to fix |
| **D3** | 1. **If the fix (or the problem) is complicated, then opening an issue means more work, because I have to explain it well enough other people will understand it. [C1]**<br>**2. If the fix is simple, then putting it in an issue makes it more likely someone will try it as their first issue. [I7]**<br><br>Regardless of those two, **if the problem impacts other work I usually open an issue anyway [I3]** (e.g. <Removed to Anonymize>).<br><br>By the way, <Removed to Anonymize> is not technical debt, it's a missing feature. Implementing it would actually make the codebase harder to work on, not easier. <Removed to Anonymize> would be a better example of tech debt. Actually I need to remove that FIXME, it's no longer relevant now that <Removed to Anonymize> exists. I guess that's another reason to put comments in code, **they're more likely to be noticed when the code is updated for an unrelated reason. [C1]** | *C1. If it provides context to the reader<br>*I3. If it spans to multiple places<br>*I7. If it is a good first issue |

| | | |
|---|---|---|
| **D4** | In my opinion, you need both. And they should be linked by automation**. There should be a 1:N relationship between bugs and todos in the codebase. When looking at a bug, there needs to be a way to refer to all the locations in the code where a TODO references that bug. The reverse direction also needs to be possible, i.e. when looking at a TODO, it should be easy to navigate to the bug teaching tracking it. [B1]** This is simple if you use the format TODO (https://github.com/issues/...), as we usually do. You caught me being lazy or forgetful in that link.<br><br>One critical element is that bugs need to receive an automated update when a TODO tracked by it is removed from the codebase. This may signify resolution of the issue or just a refactor, so there still needs to be a human in the loop to determine whether or not the issue is actually resolved.<br><br>Likewise, when a bug is closed out that references TODOs, there should either be an automated change submitted to remove those TODOs from the codebase or the bug should reopen itself until all of the TODOs it references are removed from the codebase.<br><br>I think you'd find the majority of the developers on my team agree with this assessment. We just haven't invested the time into that infrastructure yet.<br><br>Sorry if this isn't the sort of answer you were looking for. | *B1. Both Strategies |
| **D5** | I don't think I or my team have a clear "guideline" on when it is OK to leave a comment and when to create an issue. My gut feeling:<br>**1. if TODO is scoped to a module and only improves this particular piece of code - then I don't create an issue [C3]**<br>**2. if TODO is more of architectural thing, spans multiple modules, and needs input from different teams, then I'd create an issue. [I3]**<br><br>Also, I love this take: https://twitter.com/dan_abramov/status/1215838693518430210 | *C3. If it has a local scope<br>*I3. If it spans to multiple places |
| **D6** | I'm not a very structured person, so probably the issues/comments that you have seen have been done with a random criteria more than a structured way of thinking.<br><br>Said that I can answer your 2 questions.<br><br>a) **I would use them in small projects where I have control over the whole code and I revisit the code frequently. [C6] Or as part of a PR that is a Work In Progress and is going to pass through a review process (that is going to detect that TODO comments). [C1]**<br>b) I would use issues whenever **the project is big enough that you have to focus on certain part of the code, then, from my point of view, the TODO comments gets included and rarely removed. [I4] A new issue can easily get assigned and discussed in a wider environment. [I1]** | *C1. If it provides context to the reader<br>*C6. If you revisit the code frequently<br>*I4. If it requires visibility<br>*I1. If it needs discussion |
| **D7** | **I recommend both; [B1] the comment tells you where it's wrong [C1], and the issue has better discussion on it. [I1]**<br>However, if there is no place for code, the issue is the minimum. | *B1. Both Strategies<br>*C1. If it provides hints to the reader<br>*I1. If it needs discussion |

| | | |
|---|---|---|
| **D8** | I firmly believe that the best documentation for software is in code, my experience has demonstrated to me that wikis, out of tree docs, ... quickly get outdated and are useless. However, this doesn't mean that this should be the only source of documentation, high level docs, architecture, diagrams, designs, ... those things have to exist and can use wikis, docs, files, ...<br><br>To your first question, **I always recommend to document everything like TD on code ... every developer will find it always. [C1]**<br><br>To your second point, I don't think that code should be used to track issues, you need the right tool for each job, and in this case we have Github Issues to do tracking, but there are a lot of software that allows you to do this, pick your poison here :). | *C1. If it provides context to the reader |
| **D9** | I would say that often my choice to open an issue or leave a TODO isn't very scientific. Leaving a TODO is easier than writing an issue, so sometimes the decision is made out of pure laziness :) But in an ideal world, I would try to follow these guidelines:<br><br>1. **Create BOTH an issue and a TODO if possible.** Open an issue explaining the problem, and then reference the issue number from a TODO comment in a relevant part of the code. For example: <Removed to Anonymize>. **I typically use this approach if I discover an important issue in the process of writing a PR, but it's too large or difficult to fix in the current PR. [B1]**<br><br>2. **If it's not clear exactly where in the code the problem is or where a fix should be implemented, [I3]** then opening an issue without a corresponding TODO is fine. It's also perfectly fine to just open an issue even **if you know where the fix should go but you don't have an in-progress PR open for that part of the code. [I6]** In that case, you can still reference line numbers or a GitHub link in the issue description.<br><br>3. If, in the process of writing a PR, you have an idea for something that might **improve the existing code but it's not very important or urgent, then I think it's ok to just leave a TODO without opening an issue. [C2]**This can be dangerous, though, since it's easy to forget about TODOs. On my team, we typically make a point of grepping for TODOs at the end of a release cycle to make sure we didn't miss something important. I also leave TODOs without a corresponding issue number if fixing it would require more than a single issue's worth of work. In the example you gave above where I reference <Removed to Anonymize>, making the optimizer aware of latency is a big task that is on our product roadmap but may not have a corresponding GitHub issue. **TODOs can also be helpful to explain a hacky implementation so that a future reader of the code can improve it or at least understand why the original implementer made the choice that they did. [C1]** | *B1. Both Strategies<br>*C1. If it provides context to the reader<br>*C2. If it has low priority<br>*I3. If it spans to multiple places<br>*I6. If it requires medium/large effort to fix |

| | | |
|---|---|---|
| **D10** | For a), I left this as a TODO comment because **it was a small implementation detail and I didn't see it as an important issue to tackle.[C2]** If I were to create a github issue for this, it would probably be more for a general refactor of related types and then I would bundle that TODO (and related TODOs) in that issue.<br><br>For b), I opened this as a github issue above because **I saw this as something beyond just an implementation detail; it was something that would potentially affect users running end to end tests.**<br>This could have been kept as a TODO comment but I think the github issue was more appropriate because it provides a venue **to discuss potential solutions with other contributors [I1]**<br><br>Generally speaking, I think github issues are useful for **archiving larger scoped problems that impact usability or performance or code readability (just to name a few examples). [I3] Github issues are particularly useful in the <Removed to Anonymize> ecosystem because they can be tagged with "/good-first-issue" which enables new contributors to find things to work on and get familiar with the project. [I7] I think TODO comments are useful for noting smaller implementation details or "nits / nitpicks". [C1]**<br><br>It's all very context dependent, but hopefully you're able to extract something from this response to find a common pattern for which TD documentation type is best based on the TD. | *C2. If it has low priority<br>*C1. If it provides context to the reader<br>*I3. If it spans to multiple places<br>*I1. If it needs discussion<br>*I7. If it is a good first issue |
| **D11** | IMO, a TODO in code on its own isn't a record of work to do / expectation that it will be done but rather **just writing down how it might be done if it is picked up, so if I come back to this code later, there's a record of what i was thinking [C1]** w.r.t how to improve it. **An issue in the backlog is the actual "should do this thing" record, that could cause it to actually get done. [I4]** So **if I'm writing some code and think we should improve it later, I'd a) leave a todo and b) file an issue to implement what's in the TODO.[B1]** We we've done this in the past where we grep'ed through the TODOs in the PRs for a new feature and filed issues for the ones we actually wanted to come back and do. | *C1. If it provides context to the reader<br>*I4. If it requires visibility<br>*B1. Both Strategies |
| **D12** | Definitely interested in what the preferences are in the <Removed to Anonymize> code base. In my programming in general, **TODO comments are not only for documenting 'debt' but also giving myself permission to keep moving towards the overall goal despite there being problems I can't solve at the moment. [C2] And also signalling to future readers some of the bits I wasn't happy with. [C1]** | *C1. If it provides context to the reader<br>*C2. If it has low priority |
| **D13** | I write TODOs when **I have thoughts about how the code could be better [C1].** They aren't always debt. If they are debt that **has causes known problems they do likely deserve an issue too [I5].** One controversy at <Removed to Anonymize> is whether or not we want to cross reference code and issues. | *C1. If it provides context to the reader<br>*I5. If it has high priority |
| **D14** | A compelling argument against referencing issues in the code is it means that GitHub implicitly becomes part of the code base. A pragmatic reason to do it is can **unify context.** | - |

| | | |
|---|---|---|
| **D15** | a) When do you recommend documenting TD using code comments?<br><br>Generally **I will use TODO (and occasionally FIXME) in code comments as notes to myself [C1]**. As I am reading code to learn it, or when I am writing code, I will often come across things that look strange or look like they could be improved. I don't want to lose my focus on my primary objective, but I also don't want to forget about the thing I just saw (which is likely since I'm focused on something else). So I will write a short TODO to myself to remind myself later. Often I will see the notes I left for myself when I am reviewing my own code before opening a github pull request.<br><br>Sometimes I will resolve the TODOs before opening the PR, sometimes I'll create a separate branch to work on resolving it, and sometimes I will leave the comment for a later time. If I leave it I will often try to improve the wording to make it more useful to anyone else who might read it.<br><br>I would generally recommend using code comments for things that are:<br>**low priority - not likely causing any problems for users [C2]**<br>**small effort to fix - not likely to be scheduled as part of sprint planning [C4]**<br>**I think the intent of inline code comments is not to track work, but to act as hints to the reader. If I've noticed something strange about the code a TODO or FIXME can help the next reader to be less confused. [C1]** In the example you linked, the new struct is very small, so it might not be clear to readers why it exists. By adding a TODO it gives them a hint about what other fields might be added to that struct.<br><br>b) When do you recommend opening an issue?<br><br>Generally I will open an issue if I think the technical debt issue has any of the following properties:<br>**high priority - it is or could cause problems for users [I5]**<br>**medium or large effort to fix - needs to be scheduled as part of sprint planning since it will be worked on instead of other prioritized work [I6]**<br>**ambiguous - the fix may require more discussion if there are multiple options for fixing the problem [I1]**<br>**can't be fixed immediately - the change will be necessary later, but is not something that can be done immediately for some reason [I6]**<br><br>I also find that sometimes issues that start as code comments, after some time if I notice patterns of common problems in multiple places, I might choose to open an issue about it to call more attention to the problem. <Removed to Anonymize> is an example of such a case. | *C1. If it provides context to the reader<br>*C2. If it has low priority<br>*C4. If it requires small effort to fix<br>*I5. If it has high priority<br>*I1. If it needs discussion<br>*I6. If it requires medium/large effort to fix |

| | | |
|---|---|---|
| **D16** | a) When do you recommend documenting TD using code comments?<br><br>A code comment will only get resolved if someone else happens to be in the same part of the code and cares about it. Generally, you can't assume that it'll ever get fixed unless you have some specific process in place for resolving these (which I've never seen in practice). So this can only be made for really **little things that aren't worth putting into a backlog or things that you know will never get resolved anyways [C2]** (due to political or technical issues, so it's better to not clutter the backlog).<br><br>b) When do you recommend opening an issue?<br><br>**Whenever you're consciously taking on tech debt. [I4]** This should be the default and, though I've used them, code comments with "TODO" and the like should be discouraged. | *C2. If it has low priority<br>*I4. If it requires visibility |
| **D17** | **a) If it's something fairly small (which will take <1h), but that you don't want or can't spend time doing at that moment [C4]**<br>**b) When it's part of a major refactor that requires multiple steps/milestones or to outsource tasks in a more organized manner [I3]** | *C4. If it requires small effort to fix<br>*I3. If it spans to multiple places |
| **D18** | **a) when want to highlight something in the code review, particularly in cases where some limitation is explained used n other part of code [C1]**<br>**b) when I spot some problem that I don't have time to fix now [I6]; when the problem is fairly benign or simple and makes a good first issue [I7]** | *C1. If it provides context to the reader<br>*I6. If it requires medium/large effort to fix<br>*I7. If it is a good first issue |
| **D19** | IMO: To-do comments are almost always about something that you think needs refactoring at some point, but is not directly related to what you are currently trying to achieve.<br>**a) When it's something that is not a problem. [C2] Leaving a TD comment in the code can be useful when someone else bumps into the same code. [C1]**<br>**b) When something spans multiple source files, sometimes taking hours to do [I3]** | *C1. If it provides context to the reader<br>*C2. If it has low priority<br>*I3. If it spans to multiple places |
| **D20** | **a) when it's not part of the current pr but noticed when doing it. [C1]**<br>**b) when i want to encourage others to also perform some refactoring or want to document a large multi pr refactoring job. [I3]** | *C1. If it provides context to the reader<br>*I3. If it spans to multiple places |
| **D21** | a) When do you recommend documenting TD using code comments?<br><br>I would recommend TODO comments when **I want to give the reader of the code some pointer about some change we'd like to make to the code they're reading [C1]**. **A good TODO links to an issue describing the issue in more detail. [B1]**<br><br>b) When do you recommend opening an issue?<br><br>**If I expect any discussion about some piece of technical debt [I1]**, **or want to track resolution of that debt (which is nearly always)[I2]**, I want that tracked in an issue. Issues give you an opportunity to assign the issue, close the issue, refer to the issue by number, use markdown and links to refer to other discussions, etc. Issues are better than code comments in nearly every way, except that they aren't living next to the code in question. | *C1. If it provides context to the reader<br>*I1. If it needs discussion<br>*B1. Both Strategies<br>*I2. if it needs to be tracked |

| | | |
|---|---|---|
| **D22** | I use code comments when the technical debt primarily has to do with the code, such as a refactor, or **when the existence of the technical debt is important to understand the following code block. [C1]** *In your first link, it's confusing to the reader to see that most of the tests run in <Removed to Anonymize>. Adding the TODO is necessary to make the file readable.* I might also have added an issue.<br><br>**I open issues when I don't have a clear sense of how the problem I'm describing relates to the code, [I3]** or in a collaborative project if the code isn't in a section I "own". **I also open issues if it's important to document more visibly that there's a known bug or limitation with the software. [I4]** | *C1. If it provides context to the reader<br>*I3. If it spans to multiple places<br>*I4. If it requires visibility |
| **D23** | 1) When do you recommend documenting TD using code comments?<br><br>I think it's always better to document hacks / specificities / unintuitive behaviour via comments, **to help any other programmer (or your future self :) ) that would pass by the piece of code you wrote, and debug a related issue. [C1] Locality is important**, so having information such as a comment as close as possible to the problematic piece of code makes all the difference for anyone who'll be dealing with unexplained behaviour in this source file / function / code block.<br><br>**So I'd say documenting TD with code comments is important when they're about some piece of technical debt that is essentially non ideal behaviour, but self-contained, something that could certainly lead to further debugging if not highlighted by the comment. [C3]**<br><br>2) When do you recommend opening an issue?<br><br>Opening an issue is always better in the case of a community project, where t**he issue is far more visible / searchable than a code comment. [I4] It's also a way for other members of the project to put in their advice about the issue being discussed. [I1] I'd say it's less about a specific hacky code block, and more about some more abstract design decision [I3]**, or like the issue you linked, something where a decision needs to be taken by project maintainers (in this case, whether to add more dependencies to <Removed to Anonymize>). | *C1. If it provides context to the reader<br>*C3. If it has a local scope<br>*I3. If it spans to multiple places<br>*I1. If it needs discussion<br>*I4. If it requires visibility |

| | | |
|---|---|---|
| **D24** | a) When do you recommend documenting TD using comments?<br><br>I typically* try to limit commits to follow only one objective. i.e.<br>- A bug fix<br>- A refactor<br>- A new feature<br><br>Mixing and matching makes it hard to trace problems. I believe the TD comment you mentioned was introduced by a refactor, which commented on a previously existing bug/opportunity for future enhancement.<br><br>* For more trivial examples, I am sure I frequently break this. Which is why I said I try.<br><br>b) When do you recommend opening an issue?<br><br>**I typically open issues when I am looking for either volunteers to work on it [I7]**, **solicit more input [I1], or want to make others aware that it is a known issue. [I4]**<br><br>In comparing the two that you mentioned: the code comment had a viable workaround, and I think in my head I was either thinking that it was not easy for new volunteers or because of the viability did not need others to be as aware of it. **I think in my head it was more of a housekeeping chore to acknowledge it was not in an ideal state. [I4]** | *I1. If it needs discussion<br>*I7. If it is a good first issue<br>*I4. If it requires visibility |
| **D25** | The usual rule of thumb I use is that **TODOs should be for short, one-off examples of tech debt that aren't a high priority to tackle rightaway [C2] [C3].**. **Anything larger that affects multiple parts of the code base should be an issue [I3]**, as well as **anything that needs to be done with some level of certainty [I5].** | *C2. If it has low priority<br>*C3. If it has a local scope<br>*I3. If it spans to multiple places<br>*I5. If it has high priority |

| | | |
|---|---|---|
| **D26** | The problem with technical debt is often that everyone wants to get rid of it at some point, but nobody is particularly interested in actually doing the work. It is usually not easy (tech debt exists for a reason, if a better solution would have been easy, the original implementor would have chosen it), and it does not fit into the time planning of software teams who need to release visible improvements and new features.<br><br>a) I would recommend documenting TD using code comments when **the information helps to understand the code by giving additional context for either myself or a colleague in the future, but is only necessary information in this very local context. [C1][C3]** In the example you picked I didn't have the time to test if the if-condition could ever be met, but I know that I would need to return well-formed data in either case. The fact that I didn't dig deeper about how likely the error case is, is only interesting to anyone if there's a problem in that code path at some point, and then the comment will be read during the debugging / fixing process. Other than that, it simply doesn't matter.<br><br>In my opinion, **code comments are the best way to give contextual information in code [C1]** because they are most likely to survive time and be updated when the code changes. Pointing out tech debt is only one example for this.<br><br>b) I would recommend opening TD issues whenever it is decided that a certain solution is good enough to be shipped, and everyone agrees that it needs to be improved later, either in a code review with colleagues, or by a manager / tech lead. **Then the issue serves as documentation of the decision to leave the non-perfect state for now, and as a reminder of the tech debt as such. [I4] Opening tech debt issues also helps to get data about the code quality of a project that can be used to convince management to invest in either cleaning-up time, or a refactor / rewrite of the code. [I2]** In my experience tech debt issues are never just fixed in the same way as other bugs are. They are more a political /statistical instrument, if that makes sense? | *C1. If it provides context to the reader<br>*C3. If it has a local scope<br>*I4. If it requires visibility<br>*I2. if it needs to be tracked |
| **D27** | Jumping straight into it:<br>a) I typically leave code comments to indicate future work that needs to be done when the **TD in question is isolated to a particular piece of code [C3]**. I find that usually these make more sense to read in the context of the code. Sometimes **if the work item is large, it may additionally warrant an issue to be filed so that the work can be tracked as a larger unit of work [I6] [I2]**. I also find it useful to leave TODOs as an explicit note of an area of code that can be improved, but where **I did not have time to refine it (or it did not make sense to prioritize) [C2]**. This is especially useful in terms of code reviews.<br><br>b) I'll typically rely only on issues when:<br>- **there isn't a specific part of the code that the TD is addressing [I3]**<br>- **the TD needs to be tackled as a larger project involving several engineers [I6]**<br>- the TD leads to some product issue. **Filing an issue makes it easier to discuss with others and reference [I1] [I4]**.<br><br>That being said, I usually use the one that is more convenient at the time. So these are certainly not hard-and-fast rules that I always obey. | *C2. If it has low priority<br>*C3. If it has a local scope<br><br>*I3. If it spans to multiple places<br>*I4. If it requires visibility<br>*I1. If it needs discussion<br>*I6. If it requires medium/large effort to fix<br>*I2. if it needs to be tracked |

| | | |
|---|---|---|
| **D28** | I'll answer the questions as a combination, because they answers generally overlap. **In most places, I think the right thing to do is have an issue to track all places where technical debt is introduced [I2], as just a TODO in the code is likely going to get lost and not acted on. [I4]**<br><br>**Isolated code comments mentioning technical debt are useful to regain context about confusing components of code, or to guide future programmers about topics that were not necessarily actionable at the time the code was written, such as the comment you attached in the email. [C1]** | *C1. If it provides context to the reader<br>*I4. If it requires visibility<br>*I2. if it needs to be tracked |
| **D29** | As a general rule, we try to avoid TODO comments in code - the basic principle is that we track technical debt in issues, and only rarely do so in comments.<br><br>- **TD is better tracked in issues because that's where we track other things. [I2]** We could also search for TODO in our source base, but that's a separate place; it's better to have a single source of truth on tasks on our backlog.<br>- Issues keep TD out of our source good, which is generally a good thing; TD is in many cases quite low-priority and will realistically only be fixed in a long time. We don't want to have to see it all the time.<br>- However, if the TD is urgent, e.g. if I commit something which I know is problematic and must be fixed, then the opposite holds for that reason as well - **we want to stumble upon the reminder when we're in that area of the source code**. **However, in this case I'd always also open an issue (so we'd have the TD tracked in both places). [B1]**<br>-The <Removed to Anonymize> was a bit of a special case - I went through the entire codebase, annotating it for nullability with <Removed to Anonymize>. Since that's a massive undertaking across the entire codebase, I left special <Removed to Anonymize> comments in the code rather than opened issues. I could later search for these occurrences and fix them, and this was more efficient than an issue referencing each and every code site with debt.<br><br>**To summarize, we're a very issue-oriented team, and generally avoid TODO in code by default.** | *I2. if it needs to be tracked<br>*B1. Both Strategies |
| **D30** | From my perspective, code comments are useful when **there is important context for the code in question that future readers need to know [C1]**. *For example, in the link you mentioned, I am essentially trying to communicate to a future reader, "Hey, this isn't the ideal way to handle this, but there is a blocking issue. So don't treat this code as an example of the best way to do things."*<br><br>Future code readers (at least from my perspective!) will rely on existing code to understand how things work and how to implement similar things in their own code. Code comments, including TODOs, provide valuable context that informs these future developers about context surrounding existing code that they won't expect.<br><br>However, **TODOs are absolutely terrible when it comes to project management**! Ideally, I would have created another issue (or added a comment on the linked issue) mentioning this part of code which needs to be fixed when that happens.<br><br>I would say that issues are probably the most important form of communication for actually fixing the problem. **So there should be an issue created for any tech debt which absolutely needs to be fixed [I5]. This provides a means for discussion [I1]** and **project management [I2].** Then, TODOs can link back to those existing issues to provide inline context when needed. | *C1. If it provides context to the reader<br>*I5. If it has high priority<br>*I1. If it needs discussion<br>*I2. if it needs to be tracked |

| | | |
|---|---|---|
| **D31** | a) When do you recommend documenting TD using code comments?<br><br>I see two main use-cases for using code comments to document TD:<br><br>- The first one is **to give context to the reader about that a particular section is written the way it is because of technical debt. [C1]** Code with technical debt might have to perform be redundant work (such as in the example, where the same data is exposed in two different fields to support older clients), perform certain data conversions or implement special cases separately to address shortcomings of the implementation, or to work around bugs in external libraries or other part of the code. In all these cases it is crucial to communicate to the reader that the code is intentionally written that way.<br>- The second one is to **simply mark places in the code that need cleanup. [C3]** In contrast the the previous case, this type of comment is not used to explain anything about implementation, but rather indicates code sections that will have to be changed or deleted once the technical debt is being addressed. Using inline comments also increases the likelihood that these markers are moved around together with the code during refactoring (where as locations described in issues might become stale).<br><br>In short, code comments about technical debt are useful to communicate the impact of the technical debt on the code itself.<br><br>b) When do you recommend opening an issue?<br><br>- **Documenting technical debt in issues enables discussions of technical debt in more abstract terms, as the documentation is not tied to the code. [I3] This allows other developers to provide their input and thus collaboratively allow a team to find solutions [I1]**, which is especially needed for distributed and asynchronous development teams (which is common in open-source). **Issues can also be integrated into project planning, i.e. issues about technical debt can be assigned to certain developers, be targeted for release milestones, and can be used to track status updates of how the technical debt is addressed, especially if the technical debt is addressed outside of the tree. [I2]**<br><br>In short, issues about technical debt are useful to identify, coordinate and discuss technical debt.<br><br>-<br><br>**I'd like to point out that code comments vs issue is not necessarily an either/or decision.** In the <Removed to Anonymize> project, you will find documentation of technical debt that uses both, inline code comments as well as issues. Example:<Removed to Anonymize> | *C1. If it provides context to the reader<br>*C3. If it has a local scope<br>*I3. If it spans to multiple places<br>*I1. If it needs discussion<br>*I2. if it needs to be tracked |

| | | |
|---|---|---|
| **D32** | **I think adding comments and linking th em to a tracking issues is the best way to document technical dept and TODOs. [B1]**<br><br>If there is only a tracking issue, **it might be difficult for new contributors to find where the code is. [C1]**<br><br>If there is only code comments, **contributors can forget about them in release x. [I2]**<br><br>Also on the topic of TD - often times the worst TD is the one that is not documented and difficult to explain. The one that is 'tribal knowledge' - only the key / older maintainers know about it and it is quite complex. It could require a complete redesign and a detailed tracking issue is more important than having the code comments.<br><br>I think this is a great topic. Good luck with your research and let me know if you have more questions. | *C1. If it provides context to the reader<br>*B1. Both Strategies<br>*I2. if it needs to be tracked |
| **D33** | I always recommend using issues to track TODOs and known limitations (unless they affect users, in which case they should be documented in the user documentation).<br>I don't think we have an official guideline for the <Removed to Anonymize> project though. | - |
| **D34** | I _always_ recommend documenting tech debt in code comments. **I personally feel it's important to leave as much context as possible for the future. [C1]**<br><br>Issues are good but I find they're reviewed less often by the team, **but more likely to be discovered by external people. [I4]** | *C1. If it provides context to the reader<br>*I4. If it requires visibility |
| **D35** | (a) Documenting TD in comments should only be done if the **TD is intended to be resolved before the branch is merged [C5]**, in which case it's usually unneeded to have an issue to track it.<br>(b) Any TD that is "deferred" to after the branch is merged should have an issue created, **so the work is visible and can be assigned/tracked. [I4] [I2]**<br><br>Prioritizing TD is an issue for us too, it's not always clear how to provide value and clean up old TD at the same time. | *C5. If it will be addressed soon<br>*I4. If it requires visibility<br>*I2. if it needs to be tracked |
| **D36** | I typically reserve technical debt identified **using code comments for isolated areas of code or where the technical debt is limited to a few obvious source files [C3]**.<br><br>Writing up an issue is important if **removing the technical debt comes with greater risks, such as impacting the stability or compatibility of the software. Writing out an issue allows you, as the developer, to think more carefully about the changes. [I6]** | *C3. If it has a local scope<br>*I6. If it requires medium/large effort to fix |

| | | |
|---|---|---|
| **D37** | > a) When do you recommend documenting TD using code comments?<br><br>I always recommend documenting TD using code comments. The reasons:<br><br>- It's a kind of documentation for **code readers, so they can have a bigger picture/context about the code, and understand the code better. [C1]**<br><br>- It's easier for linter tools to find that TD and reaction, e.g, you can have CI rule that a PR with more than 5 TD is considered bad.<br><br>- **It won't be lost and everyone in team can jump in and help fixing the TD when they have time. [C6]**<br><br>> b) When do you recommend opening an issue?<br><br>In my experience, opening an issue (for a TD) is only mandatory if you can't find a feasible solution for it. For example, you can see in issue <Removed to Anonymize>, there're several solutions, and I can't decide which one is the best, **so an issue is better for other to join and discussing. [I1]** | *C1. If it provides context to the reader<br>*C6. If you revisit the code frequently<br>*I1. If it needs discussion |
| **D38** | The majority of our work happens in the <Removed to Anonymize> repo, and the TDs there are all centered around issues. There are some that exist in code, but they tend to be pretty old and haven't been touched in awhile.<br><br>The TODOs in code are mostly artifacts of a previous time when they were used more frequently. **Today if it's not in an issue, it doesn't get done. ALL our triage and prioritization is centered around issues. [I2]**<br>I would only recommend a TODO in a file **if its there to provide more context to the person who may have to touch the code next [C1]**, more so "if you're revisiting this code, consider doing X,Y,Z" Otherwise it should be an issue. | *C1. If it provides context to the reader<br>*I2. if it needs to be tracked |
| **D39** | > a) When do you recommend documenting TD using code comments?<br><br>**In any situation, where I am writing code and I know that the code I am writing has technical debt, I try to add comments noting the issues in the code, and open an issue that I include as a comment in the code. [B1]** The reason that I do this is so that if anyone ever comes across the code, **they can refer back to the issue to have any follow up discussion about potential solutions[I1]**. **Additionally, it allows us to measure at a project management level, how much technical debt we've taken on (e.g. this week, we opened 5 technical debt issues, maybe we need to slow down development) [I2]**.<br><br>> b) When do you recommend opening an issue?<br><br>The times that I open issues without leaving a comment in the code are usually when I notice an issue after the code has already been deployed. Or, it comes to my attention that there's a fair amount of technical debt that has built up without anyone mentioning it. | *I1. If it needs discussion<br>*I2. if it needs to be tracked<br>*B1. Both Strategies |

| | | |
|---|---|---|
| **D40** | Our policy is "no TODOs in code." They should always be issues instead. Sometimes we add TODOs in development branches, but we try to remove them before merging the development work to main. **It's fine to have a comment that mentions that something is incomplete and links to the issue [B1]**, but not a "TODO: such and such", especially if it's not paired with an issue. Of course, we don't always follow this policy, but it's more because we were lazy or forgot than for some more principled reason.<br><br>The official policy is at the bottom of this section:<br><Removed to Anonymize><br><br>The comment in <Removed to Anonymize> I wouldn't personally describe as a TODO at all. It's explaining how that code works and why, not suggesting that something should change. | *B1. Both Strategies |
| **D41** | a) When **there is a clear place in the code where the issue needs to be addressed [C3]**. **I always prefer that the comment include a link to the issue as well. [B1]**<br>b) Always | *C3. If it has a local scope<br>*B1. Both Strategies |
| **D42** | **Tasks in comments should be reserved for coding specific concerns**. This can include API challenges (I need to use a better api once we get a version bump) or **marking some particularly ugly code that works but needs to be cleaned up eventually. [C1]**<br><br>All other tech debt should be in the issue log. | *C1. If it provides context to the reader |
| **D43** | a) When do you recommend documenting TD using code comments?<br><br>I think leaving a TODO comment is better for **giving reviewers a hint about thing you plan to do soon [C1]**, or in the case of the example a question you have for the reviewers which you might not be able to answer now or before merge, but might be useful for the next person who edits this code to consider. In that line of thinking, **TODO comments can be great for little things which other people editing the same code can pickup and handle later when they're working in that code. [C1] Most of these scenarios are very minor tech debt [C4]**, the kind that collects all over the place and would probably get buried in an issue tracker but **could be very relevant information for someone working or reading the code. [C1]**<br><br>b) When do you recommend opening an issue?<br><br>In any scenario where **tech debt is too large to ignore or requires coordination or discussion to resolve.[I3][I1] We often leave comments in our code which are just links to GitHub issues with a short description of the tech debt, [B1]** like when we skip tests which are flaky: <Removed to Anonymize> I like this style because it allows a discussion while also alerting people who are looking at the code for whatever reason where to find the discussion. | *C1. If it provides context to the reader<br>*C4. If it requires small effort to fix<br>*I3. If it spans to multiple places<br>*I1. If it needs discussion<br>*B1. Both Strategies |

| | | |
|---|---|---|
| **D44** | This is a very good question, actually quite philosophical. You can expect different answers from different <Removed to Anonymize> members, if you ask. Honestly, I need to admit there are no explicit rules written on the wall, that committers should leave a TODO in this or that way. There's no clear line drawn between these two formats. Most people do what they feel the best or the most handy. The reviewer will advise too.<br><br>The answers below only represent my personal opinions, instead of consensus from the <Removed to Anonymize> community. Although my mindset is mostly consistent, there can be exceptions where my brain is totally blurred. I don't want to make excuses for that.<br><br>---<br>When do I use code comments and when do I open an issue?<br>**I tend to open an issue when I feel like the issue is applicable to more people than who will read the piece of code and find this TODO line.[I4]** Also **I open an issue when I feel like a few lines in the comment cannot fully explain what I mean [I1]**. There are surely more than these two reasons, but on the top of my head I feel these two are the main ones.<br><br>That said, I do not open an issue **if it is relatively easy to explain what's going on in the code[C3]**. And **I do not open an issue if this piece of code will interest few people.[C4]**<br><br>As an example of the comment cannot fully explain. Imagine the lines below this TODO comment are a hack, which is due to some debt in code somewhere else. Be it the control flow or a side effect or just some legacy code I don't understand. I feel like it's hard to elaborate in just a few lines. Also consider that comments are pure text where you cannot add illustrations or many code links, PR links or external references. **It's also hard to discuss with people using code comments.[I1]** <Removed to Anonymize> is also an example of this kind. It's impossible to explain what fix is required in a few TODO lines.<br><br>As an example of the comment being irrelevant to most people. Imagine I wrote a recursive string manipulation. I know the time complexity is not optimal. But the code is not critical (it is only invoked once in a command, not frequent). So copies are just fine, except for its bad smell and giving me bad dreams at night. Your example of <Removed to Anonymize> also belongs to this category. Users barely care what underlying utility my command is using, if they provide equivalent functionality. But the one mentioned in TODO seems more maintainable. This TODO is easy to explain. This suboptimal implementation is agnostic to most people. **And it is a low priority fix.[C2]** | *C2. If it has low priority<br>*C3. If it has a local scope<br>*C4. If it requires small effort to fix<br>*I4. If it requires visibility<br>*I1. If it needs discussion |

| | | |
|---|---|---|
| **D45** | That's an interesting question. In open-source projects, I would say it's always better to open issues as **it raises awareness [I4]** and **it kind-of gives "work" to newcomers or people who want to contribute but do not know yet on what to work on. [I7]**<br>On the other hand, having some FIXME and TODO in comments **allow users that browse the source code to see things[C1]**, and depending on your editor, it can be very easy to list them. The trick here, though, is that usually a TODO or FIXME comment make sense to the one that created it.. but unless very well written, it doesn't necessarily contains enough information (compared to a well crafted issue for example).<br><br>**I personally tend to use comments initially because it's usually "things to do next" tech debt. [C5]** But sometimes, it leaks and I forget to create an issue and remove a comment for example. | *C1. If it provides context to the reader<br>*I4. If it requires visibility<br>*I7. If it is a good first issue<br>*C5. If it will be addressed soon |
| **D46** | > a) When do you recommend documenting TD using code comments?<br><br>Usually I document code with **TODO if there is some improvement that can be made in the future that probably reduces technical debt or improves the code in some way that doesn't impact the user. [C2]** I also use <Removed to Anonymize> for places where I can see a better algorithm or performance improvement being made. Say where there is a O(log N) solution to something that's being done in linear time now. Often you start with simple approaches because it gets you where you need to be faster, and often n is small.<br><br>> b) When do you recommend opening an issue?<br><br>When I can see there is either a defect that needs addressing on the code, or when there is an improvement that would have a material impact on users or may result in degrading users' experience of using the code in the future. **These things you want track and prioritise more explicitly. [I2] [I5]** | *C2. If it has low priority<br>*I2. if it needs to be tracked<br>*I5. If it has high priority |
| **D47** | The better approach is almost always to open an issue. An issue has several benefits:<br>**1. It provides a place for longer explanation of what's wrong and what needs to be improved. [I1]**<br>2. **It is searchable in the bug tracker along with other issues. [I4]**<br>3. **The work to fix it can be tagged, assigned, and so on, just like any other issue. [I2]**<br>4. **When a patch is provided which marks "Fixes" this issue, the issue is closed automatically. [I2]**<br>5. **An issue can also hold additional resources related to what needs to be fixed - pointers to tests, discussions, and so on. [I1]**<br><br>There are a few places where it makes sense to add FIXME in the code instead:<br>1. **When you're writing a lot of temporary code that you know will change in a few days, so it is pointless to open issues just to close them tomorrow. [C5]**<br>2. When you want to fix something because it is ugly, but isn't really a bug (doesn't cause incorrect operation, etc.), **so it's not really critical to fix it. [C2]** | *C2. If it has low priority<br>*C5. If it will be addressed soon<br>*I4. If it requires visibility<br>*I1. If it needs discussion<br>*I2. if it needs to be tracked |

| | | |
|---|---|---|
| **D48** | a) When do you recommend documenting TD using code comments?<br><br>I use TODO annotation following my practise in <Removed to Anonymize>,<br>**when I want to remind myself or other team members that there is something**<br>**we shall do [C1],** and **this task is typically quite simple and clear to articulate (not necessarily to implement) [C3]** in a short comment<br><br>b) When do you recommend opening an issue?<br><br>**When this tech debt is complicated enough to describe [I3]** with a good context and goal,<br>I will make a github issue. | *C1. If it provides context to the reader<br>*I3. If it spans to multiple places<br>*C3. If it has a local scope |
| **D49** | I'd say my personal rules are fluid. Generally, when I leave a TODO behind in the code, I don't expect anybody to actually do the respective thing. Or, at least, not anytime soon and not for the simple purpose of resolving the respective TODO. **The point of the comment is just code documentation, usually to highlight to the reader that something is missing, or to hint at something that might not be obvious. [C1]** In some cases, like the one you've linked, the TODO carries notes for the case that somebody does feel like resolving it.<br><br>Opening an issue is something for future work is something I do less and less, given that the number of opened issues in our repo has become astronomical. For example, today I probably wouldn't have opened the one you linked. **I'll open one if I want to insist that something gets done [I5]**, or if **I want to explicitly communicate some problem to others [I4]**- so it can take the place of an email.<br>**But nowadays I'll open issues only for very broad categories of work, where the point of the issue is to hold a conversation over years to come. [I1]** | *C1. If it provides context to the reader<br>*I4. If it requires visibility<br>*I5. If it has high priority<br>*I1. If it needs discussion |

| | | |
|---|---|---|
| **D50** | a) When do you recommend documenting TD using code comments?<br><br>First some introduction:<br><br>* Before github existed I used a TODO file in the root of the project where i kept my notes and issues in there, at least the big points, but then the code was filled with // TODO and // XXX for more specific tasks that probably make no sense outside the source.<br><br>**I find useful to drop notes in the code for your future self, as long as they keep the context.[C1]** Also, i have to say that TODO comments in the code **use to be related to small tasks or something that is working or could be done better or differently or it's not necessary to be done right now. [C4][C2]** So the next person reading this code can solve the issue or move forward.<br><br>b) When do you recommend opening an issue?<br><br>Opening tickets is usually a burocratic process that anoys many developers because it creates sometimes an unnecessary delay in the implementation, but this is usually the main reason for doing it, as **it creates a place for discussion and planification for the developers to implement it. [I1]**<br><br>**Issues are usually managed (in big projects) by non-developers so they don't need to be tied to the code and be deleted or repriorized at any time. [I2]**<br><br>Sumarizing my points:<br><br>Tickets are good for planification, issue reporting and organization/priorization<br><br>TODO comments are good for developers only and they are tied to the code and internal details that probably users and managers don't care. | *C1. If it provides context to the reader<br>*C2. If it has low priority<br>*C4. If it requires small effort to fix<br>*I1. If it needs discussion<br>*I2. if it needs to be tracked |
| **D51** | I think when time is of importance (e.g. deadline), **there should be an issue so it can be tracked and acted upon in time. [I2][I5] For larger technical debt there may be issues [I3]** as well, but for **relatively small known improvements [C4]**, I don't think that is worth it. | *C4. If it requires small effort to fix<br>*I5. If it has high priority<br>*I2. if it needs to be tracked |

| | | |
|---|---|---|
| **D52** | a) When do you recommend documenting TD using code comments?<br><br>I would not recommend, and as a rule, always avoid adding todos in production code. If I do end up adding todo's in the production code, it's because of a lapse of judgement, or it snuck in there during development and was not caught before merging into production.<br><br>b) When do you recommend opening an issue?<br><br>**If a todo is genuinely meaningful and required [I5]** (often times they are not, todos/fixmes are just nice-to-haves), open an issue. An issue for a small todo is still much preferred than a todo-statement in the source-code.<br><br>^ these are ideal case scenarios, reality of course is more messy. | *I5. If it has high priority |
| **D53** | My recommendation is to document tech-debt in comments when either **the TODO will be addressed soon (so it isn't worth the overhead of opening an issue) [C5]**, or when **the item is not important enough for an issue [C2]** (e.g. this code is inefficient, but likely fine for a while). I recommend opening an issue whenever **something is important to address but won't necessarily be handled right away. [I5]** Often we have tech-debt that has both an issue and a TODO. One important bit is that the TODOs need to be regularly looked at to make sure there was an important TODO that wasn't addressed. Some <Removed to Anonymize> teams do this on a regular cadence (i.e. near the end of the 6-month release cycle). | *C2. If it has low priority<br>*C5. If it will be addressed soon<br>*I5. If it has high priority |
| **D54** | Our team doesn’t have clear guidelines defined for how to document technical debt, so it’s just been up to the discretion of the individual developer at this point. | - |
| **D55** | I'm not sure I've personally formulated a best approach between documenting tech debt with code comments vs issues. I think the challenge with issues is that they aren't part of the code itself, and they can easily be lost and forgotten about. On the flip side, it's very challenging to provide extensive context with a code comment. One technique could be that you fill in all of the context in a dedicated commit with the code comment. **In practice I would say it's probably best to do both (document extensively in an issue, and add a TODO). [B1] The benefit of a TODO is that if someone is actively refactoring that code, they will see the TODO and consider it in their refactor, whereas that's less likely to happen in an issue. [C1]**<br><br>This does mean that TODOs tend to lean towards comments that are addressing someone who's currently working on that subset of the code.<br><br>Hope these answers help you! | *C1. If it provides context to the reader<br>*B1. Both Strategies |

| | | |
|---|---|---|
| **D56** | **TODOs are generally used for smaller tasks [C4]**, more like a mental note whereas **issues usually describe larger bugs/tasks/projects. [I6]**<br><br>Some projects use internal tools to track TODOs, and some use github bots (see https://github.com/apps/todo ) to convert TODOs into Github issues.<br><br>Depending on how large the project is, sometimes it is not sustainable to create an issue for every small task especially if there is a larger distinction between outside issue reporters (say users) and project contributors. | *C4. If it requires small effort to fix<br>*I6. If it requires medium/large effort to fix |
| **D57** | I usually use a comment for **less critical things [C2]** or **smaller suggested changes [C4]**, while open an issue for **bigger tech debt stuff that I think it will be a good idea to address in the future [I6]**. | *C2. If it has low priority<br>*C4. If it requires small effort to fix<br>*I6. If it requires medium/large effort to fix |
| **D58** | a) When do you recommend documenting TD using code comments?<br><br>**Code comments should always exist in context, but they should also refer to an issue. [B1]** Our repository has tooling that enforces that. We also have tooling that reopens issues when they are closed, but the comment is still referring to it.<br><br>b) When do you recommend opening an issue?<br><br>Everything should have an issue. **Comments exist to provide additional information and pointers in context [C1]**, and should refer to the issue. | *B1. Both Strategies<br>*C1. If it provides hints to the reader |

| | | |
|---|---|---|
| **D59** | Generally speaking, when working on a team, **filing issues helps with the prioritization and assignment of work. [I2] Writing issues makes it easier to collaborate on solutions, make clarifications, and gather information before soing the work. [I1]** They also serve as a record for past problems that may prove to be a useful resource in the future. Finally, whether you're in open source or not, you will get bug reports from outside of your team, people who can't add todos, and it's easier if all of your work is organized in one place.<br><br>However, **todos in code are useful in that they put information directly in the place where work needs to be done. [C1]** They also free you from the overhead of writing and managing issues. Personally, I find it useful to write todos while I'm actively iterating on some work, as it can help me organize my thoughts as I try out an architecture. This is especially true when stubbing out methods. If you're breaking a long process into stages, sometimes it helps to write them all out empty functions with todos. However, unless I'm working by myself, or on a personal project, I will generally only write todos as temporary placeholders. **I will either fix them or convert them to issues before summiting that code for review by others. [C5]**<br><br>Especially in corporate settings, you'll often find a aversion to or ban on writing todos in code. When I asked my coworkers, they all said never write todos ever! I disagree.<br><br>I recommend writing todos when you're not responsible to anyone else. If you're using something like git, where you can iterate offline on your own before sharing it with others, use todos as much as you need them. But once you're ready to share, clean them up, just as you might rebase before submitting a PR. **Either finish the work or if it's too big (or may need to be done by someone else) write an issue and reference it in the todo comment. [B1]**<br><br>Otherwise, it's almost always preferable to write issues working on an active codebase. The only downside is when codebases get separated from their issue tracking systems. Codes moves from versioning system to versioning system, and sometimes the issues don't follow. Just look at all of the continuations of open source projects that exist now on GitHub that once existed on SourceForge. The code migrated, but nothing else. | *C1. If it provides context to the reader<br>*C5. If it will be addressed soon<br>*I1. If it needs discussion<br>*I2. if it needs to be tracked<br>*B1. Both Strategies |