An Empirical Study on Self-Admitted Technical Debt in Modern Code Review

Yutaro Kashiwa[a], Ryoma Nishikawa[a], Yasutaka Kamei[a], Masanari Kondo[a], Emad Shihab[b], Ryosuke Sato[c], Naoyasu Ubayashi[a]

[a] *Kyushu University, Japan*
[b] *Concordia University, Canada*
[c] *The University of Tokyo, Japan*

Abstract

Technical debt is a sub-optimal state of development in projects. In particular, the type of technical debt incurred by developers themselves (e.g., comments that mean the implementation is imperfect and should be replaced with another implementation) is called self-admitted technical debt (SATD). In theory, technical debt should not be left for a long period because it accumulates more cost over time, making it more difficult to process. Accordingly, developers have traditionally conducted code reviews to find technical debt. In fact, we observe that many SATD comments are often introduced during modern code reviews (MCR) that are light-weight reviews with web applications. However, it is uncertain about the nature of SATD comments that are introduced in the review process: impact, frequency, characteristics, and triggers. Herein, this study empirically examines the relationship between SATD and MCR.

Our case study of 156,372 review records from the Qt and OpenStack systems shows that (i) review records involving SATD are about 6–7% less likely to be accepted by reviews than those without SATD; (ii) review records involving SATD tend to require two to three more revisions compared with those without SATD; (iii) 28–48% of SATD comments are introduced during code reviews; (iv) SATD during reviews works for communicating between authors and reviewers; and (v) 20% of the SATD comments are introduced due to reviewers' requests.

Keywords: Self-admitted Technical Debt, Modern Code Reviews

1. Introduction

Developers often choose an alternative approach to implement products strategically, knowing that such approaches may affect quality and performance. For example, developers occasionally have to control the severe problems caused by defects and/or implement new features under release pressure. These imperfect solutions can lead to what is often referred to as "technical debt". As with financial debt, technical debt accumulates a higher cost over time [1], making it difficult to address as time passes.

Over the decades, many studies [2][3][4] have contributed to the detection of technical debt. These studies used source code [2], coding style [3], or comments [4] to perform their detection. In recent years, an approach using source code comments became popular for technical debt detection. This approach exploits the fact that developers often place comments indicating technical debt. This type of technical debt is called self-admitted technical debt (SATD) [4][5]. Vassallo et al. [6] conducted a survey on the development projects of financial systems and found that 88% of the respondents annotated programs to tell other developers that a part is inappropriately implemented and should be fixed later. Such comments indicating SATD help the author of the change notify other developers of files/methods where technical debt resides [7][8] and benefit researchers to study how to deal with technical debt by referencing the historical fixes of SATD [9].

Several papers investigated the effect of SATD on software products [10] and process [11]. Wehaibi et al. [10] studied the effect of SATD on software quality and showed that patch-sets involving SATD (SATD changes) introduce future defects less frequently than others. However, SATD changes require more effort (e.g., larger churn and more modified files) than non-SATD changes. Palomba et al. [11] examined the relationship between refactoring and SATD. They observed that 46% of refactored classes contained an SATD comment, and one of the motivations of refactoring is to remove technical debt.

However, most of the studies assumed that the SATD comments are intro-

duced during the coding process, or they do not distinguish which process SATD comments are introduced in. In addition to the coding process, another possible occasion when SATD arises is during code reviews [6] [12]. A code review is an essential activity in software quality assurance. Reviewers check if changes are clean and if they meet their quality standards to prevent integrating inappropriate changes into their codebase [13]. Once reviewers find technical debt, they order the authors of patch-sets to annotate the technical debt in their patch-sets or may not accept the patch-set.

Since it is uncertain about the nature of SATD comments that are introduced in the review process, this study tries to reveal the nature of introductions of SATD during MCR: the impact, diffusion, triggers, and characteristics. Our work is the first to study the nature of SATD comments in the code review process. We examine the following research questions with the code review data of two projects, OpenStack and Qt:

RQ_1 **Does the Existence of SATD Impact the Code Review Process?**
Several studies [10] have reported that SATD negatively affects software quality. Thus, review records involving SATD might be less likely to be accepted or might require more modifications by reviewers.

We evaluated if review records involving SATD introductions lower the acceptance rate and increase the number of revisions. As a result of the case study, we found that review records involving SATD are 6–7% less likely to be accepted by reviews than those not involving SATD.

RQ_2 **How Often do Developers Introduce SATD Comments during Code Reviews?**
While previous work has studied SATD comments that appear in snapshots, it is uncertain whether or not they are introduced during code review. While most of the previous studies regard that the SATD comments are created by the authors of patch-sets, they may be created by the results of the discussions during code reviews.

We examined how many SATD comments are introduced in the revised patch-sets (i.e., patch-sets resubmitted after the review process started) and found that 47.7% and 27.7% of SATD comments are introduced in the revised patch-sets for OpenStack and Qt, respectively.

RQ_3 **What Are the Characteristics of SATD that Are Introduced During Code Reviews?**

It is reported that code reviews improve the quality of code [14][15][16]. Intuitively, code reviews can help find technical debt and several studies [6][12] support this intuition. However, it is not unrevealed what types of SATD can be found in the code review process.

We conducted a manual classification and observed two unique usages of SATD during reviews: "Communication" SATD comments are exploited to communicate between the author and the reviewers, which often trigger discussions; the "Work in progress" SATD is used to declare the undergoing tasks, which sometimes enable reviewers to give early feedback about a code that has already been implemented.

RQ_4 **To What Extent Are SATD Comments Introduced Because of Reviewer's Requests?**

RQ2 showed that about 28–48% of SATD comments are introduced during reviewing. They might have resulted from reviewers' requests. A previous study [17] observed some cases where the authors of patches were asked to introduce SATD comments during discussions of issues. If many introductions of SATD are due to reviewers' requests, it suggests that reviews help find technical debt. Through our manual inspection, we found that 20% of SATD comments are introduced due to reviewer requests. Commonly in OpenStack and Qt, the "Scheduling," "Work dependency," and "Problem report" are introduced due to reviewers' requests.

Paper Organization: Section 2 introduces a motivating example of our study; Section 3 indicates our case study design; Section 4 describes the moti-

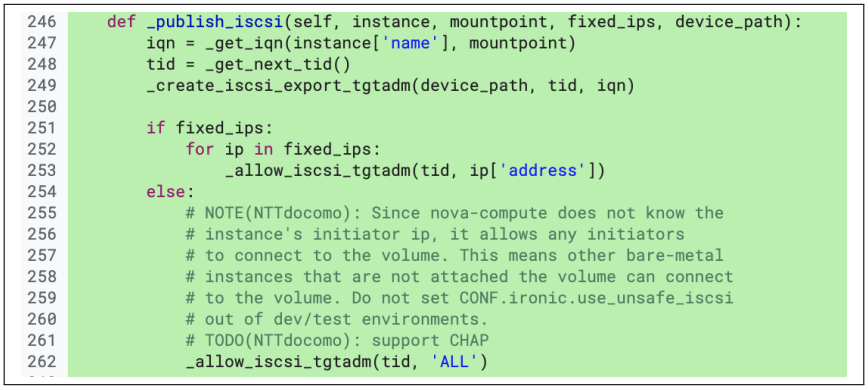
vation, approach, and results of our research questions; Section 5 discusses the broader implication of our results; Section 6 introduces related work about MCR and SATD; and Section 7 concludes our findings and discusses future work.

2. Motivating Example

Developers often choose to implement products in a non-ideal way, knowing well that such implementation might affect quality, performance, or other important factors for software. This is widely known as technical debt, and typically, should be addressed as soon as possible since it tends to "incur interest" as time passes [1][18]. Developers often note the presence of technical debt with "TODO," "FIXME," and other expressions [19] to the other developers during code implementation. This is commonly referred to as "SATD" and is an indicator of low-quality code [10].

Traditionally, developers conduct code reviews to improve the source code quality [12]. In recent years, MCRs emerged as a more instant and lightweight approach and have been widely adopted across modern software development [20]. MCR employs a web application that enables developers to remotely and asynchronously review changes (e.g., Gerrit [21], Review Board [22], Crucible [23], Phabricator [24]). Once developers make a patch-set (i.e., diff files) in their own branches and submit them into this system, reviewers can check it in the system to determine if it should be integrated into their repository. If the patch-set has problems or does not meet the project standards, reviewers may ask for revisions until the code reaches the necessary quality (or in certain cases, abandoned).

Reviewers might not accept it in the repository when they see the SATD in the patch-sets because SATD comments are often reported as one of the indicators of bad implementation [10]. We have shown an example of an SATD comment that was a cause of review rejection in Figure 1. In this figure, a method has two SATD comments, one of which warns about the method's defective behavior where specific instances connect to volumes that should not be accessed. Reviewers raise several concerns in review comments. One of these



```
246     def _publish_iscsi(self, instance, mountpoint, fixed_ips, device_path):
247         iqn = _get_iqn(instance['name'], mountpoint)
248         tid = _get_next_tid()
249         _create_iscsi_export_tgtadm(device_path, tid, iqn)
250 
251         if fixed_ips:
252             for ip in fixed_ips:
253                 _allow_iscsi_tgtadm(tid, ip['address'])
254         else:
255             # NOTE(NTTdocomo): Since nova-compute does not know the
256             # instance's initiator ip, it allows any initiators
257             # to connect to the volume. This means other bare-metal
258             # instances that are not attached the volume can connect
259             # to the volume. Do not set CONF.ironic.use_unsafe_iscsi
260             # out of dev/test environments.
261             # TODO(NTTdocomo): support CHAP
262             _allow_iscsi_tgtadm(tid, 'ALL')
```

Figure 1: Example of SATD comments that is a cause of review rejections

concerns was related to this SATD (i.e., need extra work to mount the volume), and the patch-sets were rejected as a consequence. In addition, a previous study [10] also reported that technical debt has negative effects on software quality. Hence, SATD comments might affect the decisions made in a code review. For the first research question (RQ1), we investigate if the existence of SATD comments can be a factor in code review decisions.

Shedding more light on the code review process, SATD comments are not only introduced during the submission of the first patch sets but also in the revised patch sets. The first case (i.e., where SATD comments are introduced in the first patch sets) is the case where the author of the patch-sets is made aware of the existence of technical debt and explicitly notes it in the source code as an SATD comment when they submit the source code to their reviewing system. The latter case is where the author of the patch-sets did not know that the source code contains technical debt at the time of the submission, and subsequently the author or reviewers recognized it during code reviews. For the second and third research questions (RQ2 and RQ3), we would like to investigate how many SATD comments are introduced during code reviews (i.e., in the latter case, above) and their characteristics, respectively.

During code reviews, reviewers might encourage the introduction of SATD comments. Figure 2 shows an example of a source code review with SATD

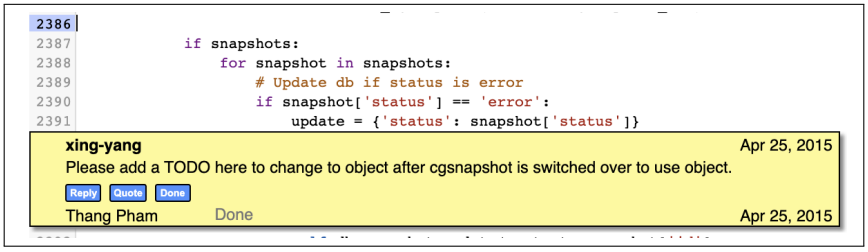


Figure 2: Example of a reviewer instruction that introduces SATD

comments introduced. The reviewer asked the author to annotate an SATD comment, and the author did so as the reviewer suggested. Consequently, the patch-set was integrated without addressing the introduced SATD comment. In this example, reviewers would be likely to make sure that the SATD is explicitly documented so that it is clear to others, and so more easily dealt with in the future. In fact, it has been reported that many SATD comments are left in snapshots [4][25]. Fucci et al. [25] reported that 0%–16% of SATD comments are introduced by other developers rather than by the owner. For the fourth research question (RQ4), we investigate what percentage of the introduced SATD comments are triggered by code reviewers.

3. Study Design

This section describes the design of our case study intended to address the four research questions. Figure 3 illustrates the overview of our data processing. In the figure, "SATD" in green characters represents that an SATD comment is introduced in the patch-set and that in gray characters shows that the introduced SATD still exists in the patch-sets. The process consists of collecting review records, detecting SATD comments, and labeling reviews.

3.1. Collecting Review Records

We selected two projects – OpenStack and Qt – that are popular and widely used in previous studies [26][27]. OpenStack and Qt have numerous sub-projects. We decided to use the same sub-projects as [26], specifically nova, glance, swift, cinder, and neutron from OpenStack, qtwebkit, qtscript, qtdeclarative, qtbase

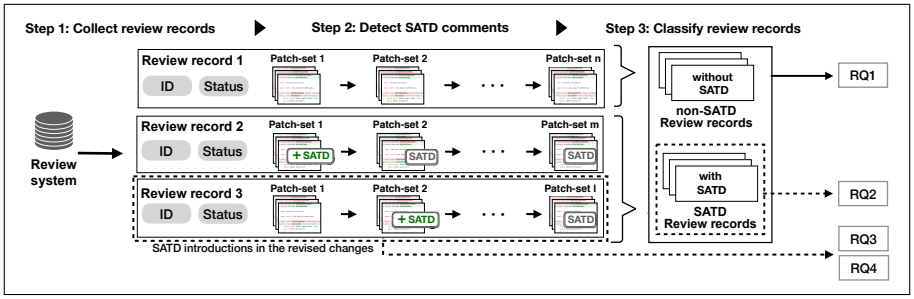


Figure 3: Overview of data processing for research questions

and the other 25 sub-projects from Qt. These projects are known for high review coverage (i.e., review most of their patch-sets) [26].

The two projects commonly use the Gerrit code review system [21], which is one of the popular MCR systems. Once a developer uploads a new patch-set and its description to the system (we name this developer as author), the system creates a new web page (i.e., a review record). On the page, reviewers assess the patch-set, give feedback, and determine if the patch-set should be integrated. If the patch-set has any issues, reviewers request a modification from the author. The author should repetitively modify and upload the revisions until the patch-sets are accepted/rejected by the reviewers.

In the system, each review record possesses an ID, a status (i.e., "OPEN," "MERGED," or "ABANDONED"), and the history of patch-sets. The status is used to determine if the patch-sets were accepted; we regard review records that have a "MERGED" status as accepted reviews. We collected them for each review record via Gerrit API [28].

3.2. Detecting SATD Comments

For every revision in each review record, we extracted all the SATD comments that are added. Then, we fed all the added comments into the SATD detection tool to identify which revision introduced SATD comments. We employed the state-of-the-art SATD detector created by Liu et al. [19] to find the SATD comments in each patch-sets (Note that we apply it to diff files, i.e., to the changed lines). The tool adopts machine learning classifiers and has been

Table 1: Our dataset summary

Project	# of reviews	# of SATD reviews	# of Non-SATD reviews	% SATD reviews
OpenStack	79,508	8,373	71,135	10.5
Qt	76,864	5,517	71,347	7.2

trained with SATD comments from eight open-source software projects. The tool takes a comment as input and provides a feedback stating whether the comment is an SATD comment or not. Finally, we identified which revision that introduces the SATD comment first and flagged it since it exists across multiple revisions.

3.3. Classifying Review Records

We classified the review records based on whether or not their patch-sets involved SATD to make two datasets: the SATD review dataset and non-SATD dataset. We label the review record as "SATD" if either of the revisions in the review has one or multiple SATD comments; else, we label the review as "non-SATD." In Figure 3, `review record 1` is labeled as "non-SATD" because no SATD comments were introduced across the revisions in this review. `Review record 2` is labeled as "SATD" because an SATD comment was introduced in `patch-set 1`. Similarly, `Review record 3` is also labeled as "SATD" because of an SATD introduction in `patch-set 2`. We aggregated the reviews that contain patch-sets where at least one SATD comment was introduced, in order to create the SATD review dataset for all RQs. We also made the non-SATD review datasets for RQ1 to compare the acceptance rate and revisions between SATD and non-SATD reviews.

Dataset Summary: Table 1 shows the total number and percentage of reviews in which at least one SATD comment was introduced either in the first patch-set or in the revised ones. We downloaded 79,508 review records from OpenStack and 76,864 from Qt. We applied the SATD detector to the reviews and found 8,373 and 5,517 review records that include at least one

SATD introduction. In about 7-10% of the reviews on average, at least one SATD comment was introduced. Note that 1,502 review records in OpenStack and 10,241 review records in Qt are excluded from the table because they are "self-reviews" (i.e., code review performed by the original coder). These have a different purpose from general-purpose code reviews, such as improving code quality and finding bugs. When only the author commented on the review record and its files (i.e., inline-comments), the record is identified as self-review.

4. Research Questions

This section describes the motivations, approaches, and results of the following research question (RQ).

RQ_1: Does the Existence of SATD Impact the Code Review Process?

Motivation. Several studies [10] reported that SATD negatively affects the software quality. Code reviews play a crucial role in "quality gate" such as preventing the integration of defective code or messy code into their repository [14][16]. Intuitively, review records involving SATD might be less likely to be accepted or might require more modifications by reviewers, while our motivating example shows that reviewers encourage authors to mention about technical debt in their source code. For the first step of this study, we hope to see the impact of SATD on the code review process.

Approach. We compare SATD reviews (i.e., the reviews in which at least one SATD comment was introduced either in the first patch-set or in the revised ones) with non-SATD reviews in order to evaluate if review records involving SATD introductions decrease the acceptance rate and increase the number of revisions. Note that we regard only the review records with a "MERGED" status as accepted reviews.

***Acceptance rate*:** For each group, we measured the acceptance rate formu-

lated as follows:

$$\begin{aligned} &\textit{Acceptance rate}_{SATD} \\ &\qquad = \frac{\textit{\# of the SATD reviews that are accepted}}{\textit{\# of all the SATD reviews}} \end{aligned}$$

$$\begin{aligned} &\textit{Acceptance rate}_{non-SATD} \\ &\qquad = \frac{\textit{\# of the non-SATD reviews that are accepted}}{\textit{\# of all the non-SATD reviews}} \end{aligned}$$

We applied the independent chi-square test (2×2) to evaluate if the two groups have a statistically significant difference ($p < 0.01$). The Chi-square test is a non-parametric test for evaluation with 2×2 matrices if two variables are independent or associated.

***Revisions*:** We measured the number of revisions for each group, which represents how many patch-sets are uploaded until acceptance/rejection. More revisions indicate that more effort was needed from the reviewers and authors to review and modify the patch-sets. We employed Mann–Whitney's U-tests ($p < 0.01$) to evaluate if a statistically significant difference exists. Mann–Whitney's U-tests are non-parametric tests of the null hypothesis that two distributions come from the same population. We also measured the effect size using the Z-score generated during the calculation of the Mann–Whitney's U-tests. The effect-size r is also non-parametric and calculated as follows:

$$r = \textit{Z-score} \;/\; \sqrt{\#of samples}$$

The effect size r varies from 0 to 1. A higher value indicates a larger difference; thus, the larger value of 0.1 (≤ 0.3) shows a small difference; the larger value of 0.3 (≤ 0.5) indicates a medium difference; and a value larger than 0.5 shows a large difference [29].

Finding 1. Review records involving SATD introductions are about 6–7% less likely to be accepted by reviewers than those not involving SATD introductions in OpenStack and Qt. Table 2 summarizes the

Table 2: Number of reviews that accepted changes with/without SATD and the acceptance rate

Project	Review	# of reviews	# of accepted reviews	Acceptance rate	p-value
OpenStack	SATD	8,373	5,690	**68.0%**	**p < 0.01**
	Non-SATD	71,135	53,266	74.9%	
Qt	SATD	5,517	4,407	**79.9%**	**p < 0.01**
	Non-SATD	71,347	61,506	86.2%	

Table 3: Number of revisions for changes with/without SATD

Project	review	# of Reviews	# of median revisions	p-Value	effect size
OpenStack	SATD	8,373	**5.00**	**p < 0.01**	0.21
	Non-SATD	71,135	2.00		
Qt	SATD	5,517	**4.00**	**p < 0.01**	0.17
	Non-SATD	71,347	2.00		

acceptance rates separated by involving SATD or not. In both OpenStack and Qt, the acceptance rate of review records involving SATD (68.0% and 79.9%, respectively) was smaller than that involving non-SATD (74.9% and 86.2%, respectively). This suggests that review records that contain patch-sets with SATD comments are less likely to be accepted. We confirmed the statistically significant difference in both projects.

Finding 2. Review records involving SATD introductions require two and three more revisions on average than those not involving SATD introductions in OpenStack and Qt, respectively. Table 3 shows the revisions with/without involving SATD. For both OpenStack and Qt, the review records involving SATD show higher median revisions (5.00 and 4.00, respectively) compared to those of non-SATD (2.00 and 2.00, respectively). We confirmed a statistically significant difference and a small effect size ($r > 0.1$) in both projects.

Answer to RQ1: The presence of SATD comments in patch-sets impacts on acceptance and revisions of review records; review records involving SATD are less about 6–7% likely to be accepted by reviews and require two and three more revisions on median average, compared with those not involving SATD.

RQ_2: How Often do Developers Introduce SATD Comments during Code Reviews?

Motivation. While previous works [10][11] have studied SATD comments that appear in snapshots, it is uncertain whether or not they are introduced during code review. Although most of the previous studies assume that the SATD comments are created by the authors of patch-sets, they might be created by the results of the discussions during code review.

If there are considerable cases where SATD comments are introduced in the revised patch-set, reviews might assist authors to find technical debt. For example, the author introduces them for the following two reasons: first, because of reviewers' requests; second, because they recognize the need to introduce them due to the discussions between reviewers and authors. We hope to know when SATD comments are introduced during reviewing. Note that RQ_2 does not distinguish whether the comments are introduced due to reviewers' requests (RQ_4 does).

Approach. We calculated what percentage of SATD comments are introduced in "initial patch-sets" or "revised patch-sets." Initial patch-sets refer to the patch-sets that start the review process, which would not have been affected by the discussions between the reviewers and authors. On the contrary, the revised patch-sets refer to the second or subsequent patch-sets, which might be improved by the discussions.

Finding 3. Developers often introduce SATD comments in the revised patch-sets (OpenStack: 47.7%; Qt: 27.7%). Table 4 shows the number and percentage of SATD introduced in the initial patch-sets and in the revised

Table 4: Number of SATD introductions during review

Project	# of All SATD introductions	# of introductions in the initial patch-sets	# of introductions in the revised patch-sets
OpenStack	21,420	11,203 (52.3%)	10,217 (47.7%)
Qt	19,662	14,208 (72.3%)	5,454 (27.7%)
Total	41,082	25,411 (61.9%)	15,671 (38.1%)

patch-sets. OpenStack developers introduced as many SATD in the revised patch-sets (47.7%) as those in the initial patch-sets (52.3%). In Qt, the number of introductions in the revised patch-sets (27.7%) was less than that in the initial patch-sets (72.3%). This suggests that code reviews may help developers find technical debt.

Answer to RQ2: 47.7% and 27.7% of the SATD introductions happened in the revised patches in OpenStack and Qt, respectively.

Additional analysis: RQ2 showed that many SATD comments are introduced in the revised patch-sets (47.7% of SATD comments in OpenStack and 27.7% in Qt). This raises another question. As with RQ1 showing that the SATD introductions lower the acceptance rate, the timing of the SATD introductions may also impact the acceptance rate. With respect to RQ1, we conducted an additional analysis. We filtered out non-SATD reviews from the datasets used in RQ1 (i.e., we used only SATD reviews.) We then split these into two groups based on the timing of the introductions: the review records where SATD comments are introduced in the initial patch-sets, and those in the non-initial patch-sets.[1]

Table 5 shows the acceptance rate when SATD comments are introduced in

[1]As there are many reviews that have both SATD comments introduced in initial patchsets and those in revised patchsets, we could not clearly separate such review records into either of them. Thus, review records that have at least one SATD comment introduced in the initial patch-sets are included in the dataset for initial patch-sets.

Table 5: Acceptance rate of reviews when SATD comments are introduced in the initial patch-sets

Project	Patch-sets	# of reviews	# of accepted reviews	Acceptance rate	p-value
OpenStack	Initial	4,510	2,898	**64.3%**	**$p < 0.01$**
	non-Initial	3,863	2,792	**72.3%**	
Qt	Initial	3,474	2,772	79.8%	$p > 0.01$
	non-Initial	2,043	1,635	80.0%	

the initial patch-sets and in the non-initial patch sets. As a result of these calculations, we observe a different result between OpenStack and Qt. Interestingly, in OpenStack, the two groups have a significant gap in the acceptance rate. The acceptance rate of SATD comments introduced in the initial patch-sets is lower than that in the non-initial patch-sets. The fact that code reviews tend to reject SATD comments in the initial patch-sets, implies that reviewers tend to prefer introducing SATD comments only when reviewers work together. RQ1 shows that in Qt, regardless of the timing, the patch-sets including SATD tend to be rejected.

Still, in both OpenStack and Qt, these acceptance rates (i.e., initial and non-initial) are lower than that in non-SATD datasets in RQ1, i.e., 74.9% in OpenStack, 86.2% in Qt (See Table 2). Again, we applied the Chi-square test with a Bonferroni correction to all pairs of distributions.[2] We found a statistically significant difference between non-SATD and Initial patch-sets groups in both OpenStack and Qt, as well as that between non-SATD and non-initial patch-sets groups only in Qt. These findings imply that SATD comments in initial patch-sets affect review decisions.

[2]Bonferroni correction is used to control for family-wise error rate

RQ_3: What Are the Characteristics of SATD that Are Introduced During Code Reviews?

Motivation. It is reported that code reviews improve the quality of code [14] [15][16]. Intuitively, code reviews may help find technical debt and also RQ2 shows that about many SATD comments are introduced during code reviews. However, it is not revealed what SATD are introduced during code reviews. In this RQ, we would like to know the characteristics of SATD comments that are introduced during code reviews.

Approach. We conduct a manual inspection to classify the introduced SATD comments. From the SATD comments that are introduced in the revised patchsets (i.e., 15,671 SATD comments), we randomly selected 375 SATD comments (OpenStack: 251 , Qt: 124), which represent a statistically significant sample with 95% confidence level and 5% confidence interval.

Next, we examined the SATD comment, its code change(s), commit message(s), and review comments. The manual inspection was performed by three authors; each SATD was independently inspected by two authors and a third one was in charge of solving conflicts. The two authors classified in the same way 80.0% of the inspected SATD comments, with a Cohen's kappa coefficient of 0.78, which demonstrates a substantial agreement [30].

Finally, we compare our categories with the categories created by a previous work [31] examining SATD comments in snapshots. Note that we decided to make our category and then compare it with the previous study's category instead of directly classifying the SATD into their categories. Because we aimed to prevent getting biased and missing the inherent characteristics of SATD during reviews.

Finding 4. SATD comments play a role in communication between authors and reviewers. Table 6 outlines the number by category for OpenStack and Qt. As a result of manual inspection, we observed that 19% of comments (47 cases in OpenStack and 26 cases in Qt) do not involve technical debt out of 375 comments. These false positives are excluded from Table

Table 6: Types of SATD that are introduced during reviews

Main cat.	Sub cat.	Description	Open Stack	Qt	Total	Sub cat. in [31]
Scheduling	Future work	This type tells the location where a new feature should be implemented in the future (i.e., after this code review).	47 (23.0%)	18 (18.4%)	65 (21.5%)	functional
	Work in progress	This type notifies what the author need to do here during reviews.	28 (13.7%)	10 (10.2%)	38 (12.6%)	-
Work dependency	Implementation	This type declares that the code with SATD will be modified after new features are developed.	21 (10.3%)	2 (2.0%)	23 (7.6%)	workaround
	Release	This type tells that the code should be modified after a future release.	13 (6.4%)	3 (3.1%)	16 (5.3%)	workaround
	Bug-Fix	This type indicates that the code should be modified after bug-fixes.	7 (3.4%)	2 (2.0%)	9 (3.0%)	workaround
	Library	This type tells that external libraries and frameworks issues block their work.	4 (2.0%)	1 (1.0%)	5 (1.7%)	workaround
	Specification	This type tells that the concrete behavior of the method is not determined.	0 (0.0%)	2 (2.0%)	2 (0.7%)	requirement
Communication	Question	This type is used to ask reviewers how the part should be implemented.	18 (8.8%)	12 (12.2%)	30 (9.9%)	-
	Suggestion	This type suggests better implementations to solve problems.	1 (0.5%)	3 (3.1%)	4 (1.3%)	-
	Review request	This type asks reviewers to deeply review specific parts of the source code.	0 (0.0%)	3 (3.1%)	3 (1.0%)	-
Problem report	Potential bug	This type points out that bugs are present in the code around the SATD.	18 (8.8%)	17 (17.3%)	35 (11.6%)	defect
	Future bug	This type points out a probability that the part may cause issues in the future.	6 (2.9%)	4 (4.1%)	10 (3.3%)	low-quality
	Maintainabillity	This type requires maintainability improvements (e.g., readability).	12 (5.9%)	2 (2.0%)	14 (4.6%)	code-smell
	Performance	This type claims the necessity of performance improvement.	8 (3.9%)	4 (4.1%)	12 (4.0%)	non-functional
Workaround	This type notifies about the reasons why the developer adopted a messy implementation.		13 (6.4%)	5 (4.1%)	18 (6.0%)	workaround
Test	Necessity	This type reports that the method does not have sufficient tests.	4 (2.0%)	3 (3.1%)	7 (2.3%)	test
	Failure	This type notices a test failure at the line where the SATD is located.	1 (0.5%)	2 (2.0%)	3 (1.0%)	test
Others	—		3	5	8	
Total			204	98	302	

6. Note that these false positives may harm the result of RQ1 while the false positive rate is lower than or equal to that in previous studies [9][16][19].

Through our manual inspection, we found six categories: "Scheduling," "Work dependency," "Problem report," "Workaround,", "Test", and "Communication." The most common category for both OpenStack and Qt is "Scheduling," which is to notice the task that should be addressed later. We describe the details for each category in Appendix.

We mapped our categories and the categories proposed by a previous study [31] to identify the unique types of SATD introduced during code reviews. The previous study categorized the SATD comments in snapshots and showed seven categories: "code debt," "design debt," "documentation debt," "defect debt," "test debt," and "requirement debt." Ours covered most of their categories while they analyzed different projects from ours[3]. For example, the most common category in the previous study "code debt" has two sub-categories: "low internal quality" and "workaround." The former was applied to our sub-category "Future bug," while the latter was applied to our category "Workaround," and sub-categories in "Work dependency". These types of SATD comments were usually used to show development tasks while pointing out a specific part of the source code. Traditionally, SATD comments are supposed to negatively affect the software development [10].

Interestingly, two unique uses of SATD comments are observed in code reviews, that is, "Communication" and "Work in progress." These were not shown in snapshots, which previous studies have investigated, because these types of SATD are deleted during reviews.

The "Communication" SATD comments are exploited to communicate between the author and reviewers. These comments can trigger a discussion of

[3]The previous study has category "documentation" which indicates outdated information (e.g., TODO left in the code but it is already fixed). The category was not observed in this study because we inspected diff files, i.e., new SATD comments developers just added at that time.

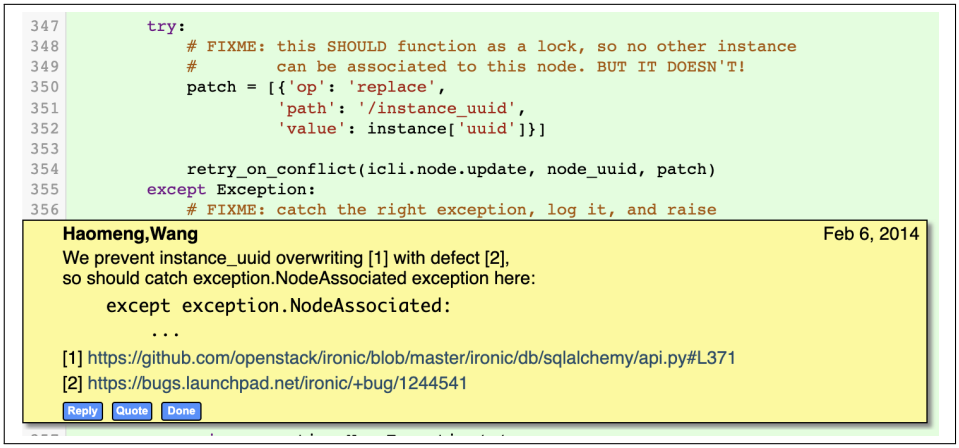


Figure 4: Reviewer feedback for an unimplemented part with the SATD of the programs

alternative options or ask questions about the specification. This type of SATD aims to facilitate development and is a unique SATD comment shown only in a code review. When the author asks for reviews, she/he suspects that the coding is the best way or the code might have a problem. Therefore, intensive reviewing of the code around such SATD comments plays an important role in improving the quality of the product. Otherwise, defects or bad implementation would be integrated into the product. The sub-categories are described in Appendix A.3.

The "Work in progress" SATD is used to declare the undergoing tasks. This type is detected because of our finer-grained analysis (i.e., we mined review systems instead of a code versioning system). If we can analyze their development activities all the time, we may detect this type of SATD comments. However, this type of SATD sometimes facilitates their development because it could enable reviewers to give early feedback about a code that has already been implemented. Figure 4 shows an example of reviewer feedback. In this way, the SATD can help their development during code reviews; thus, reviewers pay attention not to miss these SATD comments.

Answer to RQ3: We observed six types of introduced SATD during reviews: "Scheduling," "Work dependency," "Communication," "Problem report," "Workaround," and "Test." "Communication" and "Work in progress (a sub-category of Scheduling)" are unique in code reviews, which facilitate their development during code reviews.

RQ_4: To What Extent Are SATD Comments Introduced Because of Reviewer's Requests?

Motivation. RQ2 showed that many SATD comments are introduced during reviewing. If many introductions of SATD comments involve reviewers' requests, this suggests that reviews help find technical debt. Reviewers share their knowledge with the author and also with SATD; they provide the knowledge to other developers who see the code, i.e., developers might make use of reviews and SATD for knowledge sharing. In this RQ, we hope to see the extent to which SATD comments are triggered by reviewers.

Approach. For the instances that we labeled in RQ_3 except false positives (i.e., 302 SATD comments), we manually inspected the trigger of the introduced SATD comments. Two of the authors manually and independently examined the SATD comments that are categorized in RQ3 and the relevant review comments. When both authors found a review comment that orders the SATD introduction, it is labeled as a trigger. When either of the authors found it, a third author inspected it to solve the conflict. We count the triggers to show how many SATD comments are introduced by reviewers' requests.

Finding 5. About 20% of the introductions of SATD are due to reviewers' request. Table 7 shows the number of triggers that introduce SATD comments by category. Out of these 302 introduced SATD comments, 52 (25.5%) of them were triggered by reviewers in OpenStack and 8 (8.2%) were from Qt (60 cases in total). Commonly in OpenStack and Qt, the "Scheduling," "Work dependency," and "Problem report" are introduced due to reviewers' requests.

Table 7: Number of SATD comments introduced due to reviewer requests

Main category	OpenStack	Qt	Total
Workaround	5/13 (38.5%)	0/5 (0.0%)	5/18 (27.8%)
Work dependency	12/45 (26.7%)	2/10 (20.0%)	14/55 (25.5%)
Scheduling	23/75 (30.7%)	2/28 (7.1%)	25/103 (24.3%)
Problem report	9/44 (20.5%)	4/27 (14.8%)	13/71 (18.3%)
Test	1/5 (20.0%)	0/5 (0.0%)	1/10 (10.0%)
Communication	2/19 (10.5%)	0/18 (0%)	2/37 (5.4%)
Others	0/3 (0.0%)	0/5 (0.0%)	0/8 (0.0%)
Total	52/204 (25.5%)	8/98 (8.2%)	60/302 (19.9%)

On the one hand, the most common cases across all the categories in OpenStack were the SATD comments categorized in "Scheduling" (23 cases). Thus, OpenStack authors cooperated with reviewers to list-up tasks via discussion. On the other hand, Qt only has one case in "Scheduling." The most common cases involved SATD classified in "Problem Report," albeit having only four cases. Qt reviewers do not often request authors to add SATD comments.

Although the number varied across projects, 20% of SATD comments on average were triggered by reviewers, which is not a negligible number. These SATD comments are the fruits of discussions during code reviews, implying that discussion of SATD comments during the process has a real benefit. That is, code reviews can reveal hidden technical debts (i.e., not SATD), and highlighting these with SATD comments enables other developers that do not participate in

the code review to be aware of the technical debts in the source code.

Answer to RQ4: Approximately 20% of SATD comments on average were triggered by reviewers. The "Scheduling," "Work dependency," and "Problem report" are commonly introduced due to reviewers' requests in OpenStack and Qt.

5. Discussion

5.1. Is SATD introduction a real cause of rejections?

In RQ1, we observed that review records in which SATD introductions happened were about 6-7% less likely to be accepted in both OpenStack and Qt. However, it is uncertain whether SATD comments alone can be the cause of review decisions. As a result, we conducted an additional qualitative analysis to determine what percentage of SATD are the causes of rejections or acceptances. We chose 374 reviews at random from 13,890 review records that had been accepted or rejected. Then, the samples were then manually and independently examined by two authors. When conflicts occurred, they were settled by a third author.[4]

Throughout the manual inspection, we observed that only 8 review records contained the causes of review judgments due to SATD (OpenStack: 6, Qt: 2). Out of the review records, seven are rejected due to SATD comments (OpenStack: 5, Qt: 2). The number of rejected reviews due to SATD comments account for 6.5% of the total rejections (OpenStack: 6.3%, Qt: 6.9%). In terms of acceptance, we observed that only a review record in OpenStack is accepted due to SATD. In the review record, the author made a small change around a SATD comment. The majority of SATD cannot be a cause of acceptance. It is worth noting that we discovered nine review records in which authors leave

[4]The two authors classified in the same way 95.2% of the inspected SATD comments, with a Cohen's kappa coefficient of 0.48, which demonstrates a moderate agreement.

Table 8: Summary of Our Findings

#	**Findings**
Finding 1	Review records involving SATD introductions are about 6–7% less likely to be accepted by reviewers than those not involving SATD introductions in OpenStack and Qt.
Finding 2	Review records involving SATD introductions require two and three more revisions on average than those not involving SATD introductions in OpenStack and Qt, respectively.
Finding 3	Developers often introduce SATD comments in the revised patch-sets (OpenStack: 47.7%; Qt: 27.7%).
Finding 4	SATD comments play a role in communication between authors and reviewers.
Finding 5	About 20% of the introductions of SATD are due to reviewers' request

SATD comments at the end of code reviews after discussions about what they need to do in the patch-sets. While these SATD comments are not a direct cause of acceptance, they can be a compromise between the patch-authors and reviewers in order to complete reviews.

In summary, we find that while SATD is rarely the cause of acceptances, it can occasionally directly result in rejections. Several studies [10] also assert that the source code with SATD comments is frequently of poor quality. The lower acceptance rate observed in SATD reviews could be caused by the poor quality in addition to issues shown by SATD comments.

5.2. Implications

The impact, diffusion, triggers, and characteristics of SATD were investigated in this study. Our findings are presented in Table 8. Next, we examine the implications for developers and researchers in this part, mapping the findings across the research questions.

Implication 1. Code review tool developers should add a function to highlight the introduced SATD comments.

Our quantitative analysis using 156,372 review records revealed that patch-sets containing SATD comments are significantly less likely to be accepted by reviewers (Finding 1). Also, even if patch-sets with SATD comments are accepted, they require more revisions than those without SATD comments (Finding 2). These findings could be because the majority of SATD types are used to highlight issues or workaround. For example, specific categories indicating their problems (e.g., "Workaround," and "Future bugs") serve as warning signs of impending doom implementation as revealed by numerous earlier research [1][10][18]. Wehaibi et al. [10] discovered that SATD changes are more complex than non-SATD changes; in all projects and for all measurements, the effect size is either moderate or substantial (i.e., churn, the number of modified files, the number of modified directories, and change entropy).

As a result, of these effects on quality, SATD comments may have given reviewers a negative impression of the code's quality when they saw the modification. Given the reasonableness of reviewers' conclusion, developers of code review tools should add a function that highlights newly introduced SATD comments, ensuring that reviewers do not overlook issues identified by SATD comments. From the developers' perspective, if authors are unable to avoid use of SATD comments, they must properly explain why each SATD comment is included.

Implication 2. Researchers should explore approaches to suggest review comments that should be noted as SATD comments.

According to our research, 28%–48% of SATD comments are introduced during code reviews (Finding 3). Additionally, the 20% SATD comments may have been inserted by reviewers on their own initiative (Finding 5). As a result of these discoveries, code reviews work for finding hidden technical debt and notifying other developers.

Similarly, SATD also benefits code reviews. While modern code review tools can record review comments, developers need to search for them when they mod-

ify the relevant source code. Generally, the effect of review comments benefits only the patch-set author. In comparison, once SATD comments are added as a result of reviewers' comments, the review comments (i.e., SATD) remain in their source code, where they are discovered by other developers. In other words, SATD comments extend the reach of review to the other developers who did not participate in the code review because the other developers are aware of what is required to be accomplished in their source code.

For these reasons, academics should investigate methods for identifying review comments that should be noted as SATD comments, and code review tool developers should create a mechanism that allows for the easy introduction of SATD comments from the current code review tools.

Implication 3. Researchers should take into consideration that SATD is not always introduced by the patch-authors.

Again, the number of SATD comments introduced during the code review process is not negligible (Finding 3). However, past research analyzed snapshots and may have overlooked interactions between authors and reviewers. Researchers should analyze discussion data to reveal what discussions help identify technical debt. Additionally, researchers must consider the detrimental effect of code reviews on SATD investigations. For example, various studies [4][25] examine who makes SATD comments. However, the 20% SATD comments may have been introduced by reviewers' instructions (Finding 5). This implies that the patch-set authors' knowledge may be exaggerated.

Furthermore, researchers should examine another important factor throughout the code review process: SATD removals/payment [1][8][9][10]. In actuality, we discovered that a significant number of reviewers recommended ways to address SATD, as illustrated in Figure 4. Thus, by showing debt self-admittedly and reviewing it, developers would be able to pay it more efficiently than the authors could.

Implication 4. The patch-sets authors can use SATD as a communication tool.

We observed that there are two unique types of SATD categories (Finding

4). The "Communication" category is used to emphasize the section of the patch that reviewers should focus on, while the "Work In Progress" category is used to ask reviewers to check other sections of the patches in order to obtain early feedback. These categories have distinct purposes (i.e., asking reviewers to concentrate on or refrain from focusing on the code surrounding SATD) but they both serve to facilitate the review process. Thus, the SATD is not necessarily detrimental in code reviews and does not always work negatively. However, it can also be used positively. Maldonado et al.[8] also indicated, as highlighted in their survey of practitioners, that while SATD can be used to obtain feedback, it is not often employed.

While standard SATD comments stating errors may have a detrimental impact on the outcome of code reviews, particular types of SATD comments asking for reviews, queries, and early feedback can be beneficial in facilitating the review process. Patch authors should use SATD comments as a communication tool, and also researchers must investigate strategies for promoting SATD comments as a communication tool.

5.3. Threats to Validity

This section discusses the threats to the validity of our findings.

Internal Threats. First, the accuracy of the SATD detection tool should be considered. We employed a state-of-the-art SATD detector based on machine learning [19]. We measured the performance and confirmed a high precision (0.8) in RQ3, but it is imperfect. Thus, our results included several false negative/positive SATD. While the false positive rate is lower than or equal to that in previous studies [9][16][19], readers should be aware of the fact that these false positives may harm the result of RQ1.

Next, the labels might be biased because we manually inspected the reasons for SATD introduction. To mitigate the bias, the inspectors independently examined the reasons and discussed the labels afterward when they put different labels. We believe they were not biased, but we still cannot deny the bias here.

Construct Threats. To determine when an SATD comment is added, our

script compared the SATD comments of two revisions (i.e., before and after revision). If the words of two SATD comments did not match, we determined that these are different SATD comments because our program cannot distinguish if SATD comments are the same if modified. To mitigate this, we manually verified if the revision that introduced SATD comments are correct. However, we did not inspect the SATD comments in RQ1, which is a quantitative study, because it used 156,372 review records including 41,082 SATD introductions, and inspecting them manually would be tremendous. This applies to construct validity.

External Threats. Both projects in this study used Gerrit. While this is a major review system, other systems can also be used, such as GitHub pull requests. The review functions of the other systems are similar to Gerrit; thus, we believe that the effect is small.

In addition, this study investigated only two projects, namely OpenStack and Qt. Although the number of projects employed in this study is comparable to that in prior studies [27], it is not enough to generalize our findings. For future work, we will try to study projects that use different systems and increase the number of projects to generalize our findings.

6. Related Work

Our study involves aspects of both self-admited technical debt studies and modern code review studies. In this section, we will introduce work in each of these areas in turn.

6.1. Self-Admitted Technical Debt

Many studies have examined introductions, assessments, and removals of SATD. While the previous studies about the introductions of SATD are close to ours, we focused on a specific and important process: code reviews. Our work enhances the abovementioned knowledge. We have summarized them below:

Introduction of SATD comments. Potdar and Shihab [4] introduced the SATD concept and examined the causes of embedding SATD into products.

Their empirical study showed that most developers introduce SATD, but they emphasized that more experienced developers introduce more SATD. In addition, they showed that the release pressure and the complexity of components are not major factors for introducing SATD.

Vassallo et al. [6] conducted a survey on the development projects of financial systems and found that 88% of the respondents annotated programs to tell other developers that a part is inappropriately implemented and should later be fixed.

Nature of SATD comments. Maldonado et al. [5] classified SATD into five categories and showed that the majority of the SATD is related to design, accounting for 42% to 84% of the SATD in the studied systems.

Bavota et al. [31] also conducted a "differentiated replication" [32] of the study by Potdar and Shihab [4] to widen knowledge on SATD with 7,584 SATD comments from 159 software projects. They found that SATD comments increase over time. They also found that the developers who added the comments often remove them, except for when they have been removed by more experienced developers.

Li et al. [17] also classified 152 SATD items, which were found in 500 issue reports, into 8 types: architecture, build, code, defect, design, documentation, requirement, and test debt. In the study, they observed that developers asked the author of patches to introduce comments indicating SATD during discussions of issues. They claim that technical debt is identified in code review and have found some examples.

Wehaibi et al. [10] measured the effect of SATD on the software quality. They highlighted that the presence of SATD makes the code more difficult to change in the future.

Removal of SATD comments. Maldonado et al. [8] investigated SATD removal and showed that 54.4% of SATD is self-removed, which is faster than non-self-removed technical debt. Developers also mostly delete SATD after they have fixed bugs or implemented new features.

Zampetti et al. [9] examined the SATD removal in five open-source projects.

They revealed that the SATD comments were removed in one of two cases: 1) when the whole class or method is removed (25%–60% of SATD); and 2) when the method is modified (33%–63%). In addition, they classified the SATD removals into 12 categories and found that the SATD comments were often removed while improving/adding features.

Maipradit et al. [33][34] also examined the SATD removals and identified "on-hold SATD" which indicates a condition to remove itself. In the literature, they claimed that these SATD can be automatically managed and then developed a prediction tool that can detect on-hold SATD with an AUC of 0.98.

6.2. Code Reviews

While a considerable amount of studies have focused on the MCR to remove defects, to the best of our knowledge, no study has yet investigated the MCR to remove technical debt, even though technical debt will cause defects or other obstacles. Our work also has the same direction as the studies investigating the effect on reviews, but we studied a different subject. In the following, we introduce the previous studies, focusing on different aspects, including the benefits of code reviews and effects on code reviews.

Benefits of Code Reviews. Bacchelli and Bird [14] performed interviews with Microsoft developers and asked them to complete a survey. This was conducted to investigate the motivation for the MCR and its outcome. They revealed that while the first and second most common motivations are finding defects and improving the code quality, the next most common motivation is the creation of alternative approaches.

McIntosh et al. [20] investigated the relationship between defects and code review. They revealed a negative correlation between software quality and review coverage, i.e., the smaller the percentage of lines the developers reviewed in each change, the more defects the developers missed in the changes. In addition, they showed that fewer participants or discussions result in defects. Subsequently, Shimagaki et al. [35] conducted a study replicating that of McIntosh et al. [20] using data from Sony Mobile to measure the effect of code review

practices. They showed that no correlation exists between defect-proneness and the measures proposed in the previous study [20] (i.e., review coverage and participation).

Bavota and Russo [16] studied whether code reviews have a relationship with defect inducement or software quality. Un-reviewed changes were over two times more likely to introduce defects than reviewed changes. Thongtanunam et al. [36] also studied the MCR practices in defective and clean source code files and found that defective files tend to be reviewed less rigorously than clean files. Moreover, Morales et al. [15] investigated the effect of the code review practice on the software design quality. They used seven different types of anti-patterns as a proxy measure and observed that anti-patterns are likely to occur in software components with a low review coverage or low review participation.

Effects on Reviews. Rigby et al. [37] conducted a case study to investigated peer-reviews in e-mails from the Apache project. They showed that the developers are more likely to only accept patch-sets with all three properties: small, independent, and complete. Weissgerber et al. [38] analyzed the accepted patch-sets and found that the patch-set size influences its acceptance. Small patch-sets (less than four lines) tend to be accepted, accounting for 50% of all the accepted patch-sets. In addition, only 11% of the large patch-sets (more than 25 lines) are accepted. As for revisions, Beller et al. [39] showed that a greater number of modified files and a higher code churn incur more revisions, while metrics about reviewers do not affect it.

Meanwhile, Baysal et al. [40] examined the acceptance rate from a different view, that is, non-technical factors. They showed that the author's experience affects the acceptance of patch-sets: less experienced developers tend to receive rejections. Kononenko et al. [41] conducted a survey of 88 Mozilla core developers to examine the factors of review decisions and code assessments. In the survey, developers said that in addition to code quality, presence, and quality of tests, developer personality also affects review acceptances.

Zampetti et al. [42] investigated how the results of continuous integration

(CI) affect pull request acceptance. They revealed that the pull requests that passed CI builds are 1.5 times more likely to be accepted than those that fail.

7. Conclusion

Developers often notify other developers of the presence of technical debt with "TODO," "FIXME," and other expressions during code implementation. This is called the SATD, which is also an indicator of a low-quality code.

Traditionally, developers conduct code reviews to improve the source code quality. In recent years, MCRs have emerged as a more instant and light-weight approach widely adopted across modern software development. While inspecting source code during reviews, reviewers would recognize the presence of TD or SATD in the source code because many SATD comments exist in snapshots. When reviewers find technical debt (not admitted by the authors) or SATD in the source code, they take actions to handle the SATD when they meet it or the technical debt (e.g., reviewers would ask the author to add them).

However, to the best of our knowledge, none of the previous studies have yet investigated SATD comments during reviews. Therefore, how reviewers see SATD comments (e.g., reviewers tend to reject patch-sets with SATD), why, and how often they are introduced are unclear.

We conducted a study aiming to understand the effect of SATD comments on accepting and revising patch-sets and the practice of introducing SATD in code reviews. As a quantitative study, we collected 156,372 review records from OpenStack and Qt. We showed that (i) review records involving SATD are about 6–7% less likely to be accepted by reviews compared to those without SATD; (ii) review records involving SATD tend to require 2–3 more revisions compared with those without SATD; and (iii) 28–48% of SATD comments are introduced during code reviews. As a qualitative study, we also manually inspected 361 SATD comments and their reviews and showed that: (iv) SATD during reviews works for communicating between authors and reviewers; and (v) approximately 20% of the SATD comments are introduced due to reviewer requests.

For future work, we are planning to collect more review records from different

projects and create a prediction model to find the locations where SATD should be noted instead of reviewers.

Acknowledgement

We gratefully acknowledge the financial support of JSPS and SNSF for the project "SENSOR" (No. JPJSJRP20191502), and JSPS for the KAKENHI grants (JP21H04877, JP21K17725).

References

[1] Y. Kamei, E. da S. Maldonado, E. Shihab, N. Ubayashi, Using analytics to quantify interest of self-admitted technical debt, in: Proceedings of the Joint of the 4th International Workshop on Quantitative Approaches to Software Quality and 1st International Workshop on Technical Debt Analytics (TDA), 2016, pp. 68–71.

[2] B. Curtis, J. Sappidi, A. Szynkarski, Estimating the size, cost, and types of technical debt, in: Proceedings of the Third International Workshop on Managing Technical Debt (MTD), 2012, pp. 49–53.

[3] F. A. Fontana, V. Ferme, S. Spinelli, Investigating the impact of code smells debt on quality code evaluation, in: Proceedings of the Third International Workshop on Managing Technical Debt (MTD), 2012, pp. 15–22.

[4] A. Potdar, E. Shihab, An exploratory study on self-admitted technical debt, in: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 91–100.

[5] E. da S. Maldonado, E. Shihab, Detecting and quantifying different types of self-admitted technical debt, in: Proceedings of the 7th IEEE International Workshop on Managing Technical Debt (MTD), 2015, pp. 9–15.

[6] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. D. Penta, A. Zaidman, Continuous delivery practices in a large financial organization, in: Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 519–528.

[7] Y. Miyake, S. Amasaki, H. Aman, T. Yokogawa, A replicated study on relationship between code quality and method comments, in: Proceedings of the 4th Applied Computing and Information Technology (ACIT), 2017, pp. 17–30.

[8] E. da S. Maldonado, R. Abdalkareem, E. Shihab, A. Serebrenik, An empirical study on the removal of self-admitted technical debt, in: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 238–248.

[9] F. Zampetti, A. Serebrenik, M. D. Penta, Was self-admitted technical debt removal a real removal?: an in-depth perspective, in: Proceedings of the 15th International Conference on Mining Software Repositories (MSR), 2018, pp. 526–536.

[10] S. Wehaibi, E. Shihab, L. Guerrouj, Examining the impact of self-admitted technical debt on software quality, in: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 179–188.

[11] F. Palomba, A. Zaidman, R. Oliveto, A. D. Lucia, An exploratory study on the relationship between changes and refactoring, in: Proceedings of the 25th International Conference on Program Comprehension (ICPC), 2017, pp. 176–185.

[12] J. Yli-Huumo, A. Maglyas, K. Smolander, How do software development teams manage technical debt? - an empirical study, Journal of Systems and Software 120 (2016) 195–218.

[13] S. McIntosh, Y. Kamei, B. Adams, A. E. Hassan, An empirical study of the impact of modern code review practices on software quality, Empirical Software Engineering 21 (5) (2016) 2146–2189.

[14] A. Bacchelli, C. Bird, Expectations, outcomes, and challenges of modern

code review, in: Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 712–721.

[15] R. Morales, S. McIntosh, F. Khomh, Do code review practices impact design quality? A case study of the qt, vtk, and ITK projects, in: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 171–180.

[16] G. Bavota, B. Russo, Four eyes are better than two: On the impact of code reviews on software quality, in: Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 81–90.

[17] Y. Li, M. Soliman, P. Avgeriou, Identification and remediation of self-admitted technical debt in issue trackers, in: Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2020, pp. 495–503.

[18] Z. Li, Q. Yu, P. Liang, R. Mo, C. Yang, Interest of defect technical debt: An exploratory study on apache projects, in: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 629–639.

[19] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, S. Li, SATD detector: a text-mining-based self-admitted technical debt detection tool, in: Proceedings of the 40th International Conference on Software Engineering (ICSE): Companion Proceedings, 2018, pp. 9–12.

[20] S. McIntosh, Y. Kamei, B. Adams, A. E. Hassan, The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects, in: Proceedings of the 11th Working Conference on Mining Software Repositories (MSR), 2014, pp. 192–201.

[21] Gerrit. https://www.gerritcodereview.com/ [online].

[22] Reviewboard. https://www.reviewboard.org/ [online].

[23] Crucible. https://www.atlassian.com/software/crucible [online].

[24] Phabricator. https://www.phacility.com/phabricator/ [online].

[25] G. Fucci, F. Zampetti, A. Serebrenik, M. D. Penta, Who (self) admits technical debt?, in: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 672–676.

[26] S. McIntosh, Y. Kamei, Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction, IEEE Transactions on Software Engineering 44 (5) (2018) 412–428.

[27] P. Thongtanunam, S. McIntosh, A. E. Hassan, H. Iida, Review participation in modern code review - an empirical study of the android, qt, and openstack projects, Empirical Software Engineering 22 (2) (2017) 768–817.

[28] Gerrit API. https://gerrit-review.googlesource.com/documentation/rest-api.html [online].

[29] C. Hugh, Research methods and statistics in psychology.

[30] J. R. Landis, G. G. Koch, The measurement of observer agreement for categorical data, Biometrics 33 (1977) 159–174.

[31] G. Bavota, B. Russo, A large-scale empirical study on self-admitted technical debt, in: Proceedings of the 13th International Conference on Mining Software Repositories (MSR), 2016, pp. 315–326.

[32] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, Experimentation in Software Engineering - An Introduction, Vol. 6 of The Journal of The Kluwer International Series in Software Engineering, 2000.

[33] R. Maipradit, C. Treude, H. Hata, K. Matsumoto, Wait for it: identifying "on-hold" self-admitted technical debt, Empirical Software Engineering 25 (5) (2020) 3770–3798.

[34] R. Maipradit, B. Lin, C. Nagy, G. Bavota, M. Lanza, H. Hata, K. Matsumoto, Automated identification of on-hold self-admitted technical debt, in: Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 54–64.

[35] J. Shimagaki, Y. Kamei, S. McIntosh, A. E. Hassan, N. Ubayashi, A study of the quality-impacting practices of modern code review at sony mobile, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 212–221.

[36] P. Thongtanunam, S. McIntosh, A. E. Hassan, H. Iida, Investigating code review practices in defective files: An empirical study of the qt system, in: Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR), 2015, pp. 168–179.

[37] P. C. Rigby, D. M. Germán, M. D. Storey, Open source software peer review practices: a case study of the apache server, in: Proceedings of the 30th International Conference on Software Engineering (ICSE), 2008, pp. 541–550.

[38] P. Weißgerber, D. Neu, S. Diehl, Small patches get in!, in: Proceedings of the 5th International Working Conference on Mining Software Repositories (MSR), 2008, pp. 67–76.

[39] M. Beller, A. Bacchelli, A. Zaidman, E. Jürgens, Modern code reviews in open-source projects: which problems do they fix?, in: Proceedings of the 11th Working Conference on Mining Software Repositories (MSR), 2014, pp. 202–211.

[40] O. Baysal, O. Kononenko, R. Holmes, M. W. Godfrey, Investigating technical and non-technical factors influencing modern code review, Empirical Software Engineering 21 (3) (2016) 932–959.

[41] O. Kononenko, O. Baysal, M. W. Godfrey, Code review quality: how de-

velopers see it, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 1028–1038.

[42] F. Zampetti, G. Bavota, G. Canfora, M. D. Penta, A study on the interplay between pull request review and continuous integration builds, in: Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, pp. 38–48.

[43] X. Han, A. Tahir, P. Liang, S. Counsell, Y. Luo, Understanding code smell detection via code review: A study of the openstack community, in: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC), 2021, pp. 323–334.

Appendix A. Examples of Introduced SATD Comments

A.1. Scheduling

This type of SATD conveys to the other developers that the task is not completed yet. We found 103 cases belonging to this type: 75 in OpenStack and 28 for Qt. We then categorized this type into two sub-categories "Future work" and "Work in progress," based on whether the task was completed or not during the review.

Future work: This type of SATD is used to notify developers of the location where a new feature should be implemented.

Snippet 1 shows an example of the SATD categorized into this type. This SATD comment was added when fixing a bug that failed during disk resizing operations. The author addressed the problem only for common disks; hence, a reviewer asked the author to place the SATD in the source code to indicate that another disk type should be supported.

Snippet 1: Example of the SATD additions categorized into "Future work"

```
1 +   # Resize the disk (if larger)
2 +   old_root_gb = instance.system_metadata['old_instance_type_root_gb']
3 +   if instance['root_gb'] > int(old_root_gb):
4 +       ...
5 +   # TODO(ericwb): add extend for ephemeral disk
```

Work in progress: Developers use this type of SATD to notify about what they implement while they increment the program by multiple commits during reviews. This allows the authors to receive early feedback, even though the implementation is not completed. We labeled this type of SATD comments when we found that the task written in the SATD comment was completed during the review.

Snippet 2 shows a "Work in progress" example of the added SATD comment in the source code. At the beginning of the code review, an author implemented a new method and inserted the SATD by the same commit.

This review lasts 29 revisions; therefore, the exception was implemented at the final revision (Snippet 3).

Snippet 2: Example of the SATD additions categorized into "Work in Progress"

```
+ def spawn(self, context, instance, image_meta, injected_files,
+   admin_password, network_info=None, block_device_info=None):
+       ...
+   except Exception:
+       # FIXME: catch the right exception, log it, and raise
+       raise
```

Snippet 3: The Snippet of later patch for Snippet 2

```
  def spawn(self, context, instance, image_meta, injected_files,
    admin_password, network_info=None, block_device_info=None):
        ...
-   except Exception:
-       # FIXME: catch the right exception, log it, and raise
-       raise
+   except (ironic_exception.HTTPBadRequest, MaximumRetriesReached):
+       msg = _("Unable to set instance UUID for node %s") % node_uuid
+       LOG.error(msg)
+       raise exception.NovaException(msg)
```

A.2. Work dependency

This type of SATD is created when developers cannot resolve it because other issues block the progress. Therefore, they need to wait until the relevant issues are resolved. This type of SATD comments is also called "On-hold SATD" [33][34]. We found 55 cases classified into four sub-categories based on what developers wait for: "Implementation," "Release," "Bug-Fix," and "Library."

Implementation: Developers use this type of SATD to state that the author plans to use an implementation under development.

We found 23 cases for this: 21 from OpenStack and 2 from Qt. The SATD in Snippet 4 asks developers to uncomment the next lines to enable them after the developers completed an implementation to obtain the Port object.

Snippet 4: Example of the SATD additions categorized into “Implementation”

```
+ class Network(base.NeutronDbObject):
+   ...
+   fields = {
+       ...
+       # synthetic fields:
+       'subnets': obj_fields.ListOfObjectsField('Subnet', nullable=True),
+       # TODO(korzen): get Port object when implementation is ready
+       # 'ports': obj_fields.ListOfObjectsField('Port')),
+       'dhcp_agents': obj_fields.ObjectField('NetworkDhcpAgentBinding',
+                                nullable=True)
+       ...
```

Release: The SATD in the “Release” category indicates that the relevant code should be modified after a future release. We found 13 cases in OpenStack and 3 cases in Qt. We show an example in Snippet 5, illustrating that the function needs to be removed after the project released a version. Like this example, some SATD comments seem to be just tasks and would not be real debts that accumulate interest. Still, they may lower the readability of the source code. Such unused code (i.e., “dead code”) can be categorized into one of code smells [43].

Snippet 5: Example of the SATD additions categorized into “Release”

```
+ // ### Qt6: FIXME: Remove this function. It is only there since for binary
+ // compatibility for applications built with Qt 5.1 using qtmain.lib which calls it.
+ // In Qt 5.2, qtmain.lib was changed to use CommandLineToArgvW() without calling into
  Qt5Core.
  Q_CORE_EXPORT
  void qWinMain(HINSTANCE instance, HINSTANCE prevInstance, LPSTR cmdParam,
       int cmdShow, int &argc, QVector<char *> &argv)
  ...
```

Bug-Fix: This type of SATD indicates that the relevant code should be modified after bug-fixes. We found seven cases in OpenStack and two cases in Qt. Snippet 6 shows that an author asks developers to remove an exception in the code after Bug #1413142 is fixed.

Snippet 6: Example of the SATD additions categorized into “Bug-fix”

```
+   # FIXME(sahid): At this step we probably want to break the
+   # process if something wrong happens however our CI
+   # provides a bad configuration for libguestfs reported in
+   # the bug lp#1413142. When resolved we should remove this
+   # except to let the error to Pbe propagated.
```

```
-   LOG.debug('Unable to mount image %(image)s with '
+   LOG.warn(_LW('Unable to mount image %(image)s with '
                 'error %(error)s. Cannot resize.'),
             {'image': image, 'error': e})
    ...
```

Library: This type of SATD is made when libraries and frameworks block their work or require a superfluous processing. We obtained four cases in OpenStack and one case in Qt. The SATD example in Snippet 7 shows that some statements are required for MySQL 5.5 supports but they are going to be removed after the project decided to drop the support of MySQL 5.5.

Snippet 7: Example of the SATD additions categorized into "Library"

```
+ def _worker_set_updated_at_field(values):
+     # TODO(geguileo): Once we drop support for MySQL 5.5 we can simplify this
+     # method.
+     updated_at = values.get('updated_at', timeutils.utcnow())
+     if isinstance(updated_at, six.string_types):
+         return
+     if not DB_SUPPORTS_SUBSECOND_RESOLUTION:
+         updated_at = updated_at.replace(microsecond=0)
+     values['updated_at'] = updated_at
```

Specification: This type of SATD is created when the concrete behavior of the method is not determined at the time of implementation. We found two cases only in Qt. In Snippet 8, an author added an IF-statement to block the subsequent statements from executing and introduced an SATD comment that asks other developers to enable the subsequent statements to run by removing the IF-statement after the behavior of the method was decided.

Snippet 8: Example of the SATD additions categorized into "Specification"

```
 void QCocoaWindow::handleGeometryChange()
 {
-    // Don't send geometry change event to Qt unless it's ready to handle events
+    // Prevent geometry change during initialization, as that will result
+    // in a resize event, and Qt expects those to come after the show event.
+    // FIXME: Remove once we've clarified the Qt behavior for this.
-    if (m_inConstructor)
+    if (!m_initialized)
        return;
```

A.3. Communication

This type of SATD comments is introduced to make an opportunity to discuss problems or concerns with other developers. We observed 37 cases in total. These are classified into four sub-categories below.

Question: Developers use this type of SATD to ask about a specific part of programs. We obtained 30 cases: 18 cases from OpenStack and 12 from Qt.

Snippet 9 illustrates an example of SATD. The author asked about an uncommon implementation written by another developer via an SATD comment but eliminated the comment with no later discussions.

Snippet 9: Example of the SATD additions categorized into "Question"

```
1   def __init__(...):
2     ...
3     # Note: you probably want to call MimePutter.connect() instead of
4     # instantiating one of these directly.
5 +   # XXX And the reason you're not passing chunked to __init__() is...?
6     self.chunked = True # MIME requests always send chunked body
7     ...
```

Suggestion: Developers suggest better approaches to improve efficiency, maintainability, or other aspects of quality. We found three cases in Qt.

In the example of SATD shown in Snippet 10, a developer added some statements in a loop but he/she claims that the loop is not efficient and that it should be replaced with another loop that she/he proposes.

Snippet 10: Example of the SATD additions categorized into "Suggestion"

```
1 +   // ### could probably get better limit by looping over sorted list and counting down
    on ending edges
2 +   if ((v->flags & (LineBeforeStarts|LineAfterStarts))
3 +       && !(v->flags & (LineAfterEnds|LineBeforeEnds)))
4 +       *maxActiveEdges += 2;
```

Review request: Developers sometimes request reviewers (or themselves) to deeply review specific parts of the source code. The requests are usually made on a Gerrit, page but developers perhaps wanted reviewers to

review them during the implementation. Snippet 11 shows an example of an SATD comment that asks reviewers to check the logic. A reviewer inspected it later and discussed the logic.

Snippet 11: Example of the SATD additions categorized into "Review request"

```
1     int yday = dayOfYear();
2     int wday = dayOfWeek();
3 +   // TODO: Check this logic.
4     int week = (yday - wday + 10) / 7;
```

A.4. Problem report

This type of introduced SATD points out various types of problems. We obtained 71 cases, of which 44 are from OpenStack and 27 are from Qt. They were then further classified into four sub-categories.

Potential bug: This type of SATD points out that bugs are present in the code around SATD. We found 35 cases: 18 in OpenStack and 17 in Qt. Snippet 12 shows an example of this category. The SATD indicates the presence of a bug, but the developers do not figure out the cause. This SATD was added after the "if statement" was modified because a reviewer asked the author to write the reason why the "if statement" was modified.

Snippet 12: Example of the SATD additions categorized into "Potential bug"

```
1 - #if 1
2 + #if 0 // TODO: This code appears to crash seldomly in presence of tilt. Requires
    further investigation
3     std::vector<std::vector<c2t::Point>> clipperPoints;
4     ...
```

Future bug: This type of SATD suggests a probability that the implementation might become the cause of bugs by modifying the code. This category is close to the category "Future bug," but the implementation does not have bugs (or the symptoms have not appeared) at the moment. OpenStack shows six cases, while Qt has four. Snippet 13 illustrates an

example that prompts developers to learn about the method specification because modifying without understanding results in bugs.

Snippet 13: Example of the SATD additions categorized into "Future bug"

```
- // Does not work for POST/PUT!
+ // NOTE: MiniHttpServer has a very limited support of PUT/POST requests! Make
+ // sure you understand the server's code before PUTting/POSTing data (and
+ // probably you'll have to update the logic).
  class MiniHttpServer: public QTcpServer
  {
    ...
```

Maintainability: This type of SATD requires maintainability improvements. For example, the program must be capsulated, generic, and simplified. We found 12 and 2 cases in OpenStack and Qt, respectively. Snippet 14 shows an example that the argument of the list should be a generic class instead of a concrete class.

Snippet 14: Example of the SATD additions categorized into "Maintainability"

```
+   # FIXME(comstud): Make more generic later. Finish 'volume' and
+   # 'network' service code
+   args = list(method_info['method_args'])
```

Performance: This type of SATD claims that there are rooms for performance improvement or indicates performance problems. We found eight cases in OpenStack and four in Qt. Snippet 15 contains an SATD comment that requests using a cache in their method.

Snippet 15: Example of the SATD additions categorized into "Performance"

```
+ def update_provider_enabled(self, context, rp_uuid, enabled):
+   ...
+    # Get the current traits (and generation) for the provider.
+   # TODO(mriedem): Leverage the ProviderTree cache in get_provider_traits
+   trait_info = self.reportclient.get_provider_traits(context, rp_uuid)
```

A.5. Workaround

This type of SATD is introduced to notify of reasons why the developer has adopted a messy implementation. Developers often avoid several problems caused by such implementations.

18 cases were classified into this class: 13 in OpenStack and 5 in Qt. Snippet 16 is located in a test code, and the author justified that an error code is appropriate. The error code is too general to understand the cause. However, a more specific code may miss the cause of the problem because this problem happens by one of several causes. Therefore, the author notified other developers of the reason why she/he employs the error code. After that, one of the reviewers agreed with using the code. The patch-sets are finally merged into their repository.

Snippet 16: Example of the SATD additions categorized into "Workaround"

```
+   # I know that HTTP 500 is harsh code but I think this conflict case
+   # signals either a serious db inconsistency or a bug in nova's
+   # claim code.
+   self.assertEqual(500, exception.response.status_code)
```

A.6. Test

This type of SATD identifies tasks or problems with testing. We found ten cases, which were further classified into two sub-categories of "Necessity" and "Failure."

Necessity: This type reports that the method does not have sufficient tests. We found four cases in OpenStack and three in Qt.

In Snippet 17, the SATD stated that the developers need to test the methods. Finally, the test was added, and the SATD comment was deleted.

Snippet 17: Example of the SATD additions categorized into "Necessity"

```
def get_rule_types(self, context, filters=None, fields=None,
     sorts=None, limit=None,
     marker=None, page_reverse=False):
-   pass
+   #TODO(QoS): API test needed
+   return self.core_plugin.supported_qos_rule_types
```

Failure: This type notifies developers that a test fails at the line where the SATD is located. We obtained a case in OpenStack and two cases in Qt. Snippet 18 depicts one of the examples. The SATD comment indicates that a test incurs UnicodeEncodeError, but the author cannot specify the cause. The SATD comment and its relevant code end up being integrated without changes.

Snippet 18: Example of the SATD additions categorized into "Failure"

```
+   except UnicodeEncodeError:
+       self.logger.warn("xml field %s value %s \
+           can't be utf-8 decoded." % (field, orig))
+       # TODO - find out why this happens in functests
+       #return None
```