コードレビューを通じたSelf-Admitted Technical Debtの追加・削除に関する実証的研究

西川 諒真[1,a)]　亀井 靖高[1,b)]　佐藤 亮介[1,c)]　鵜林 尚靖[1,d)]

概要：Self-Admitted Technical Debt（SATD）とはプロジェクトにおける最適ではない開発状態を示す技術的負債の中の一種であり，開発者がコメントを用いてソースコード上へ自発的に言及した技術的負債のことを指す．技術的負債は長く放置すると処理にかかるコストが増大する傾向を持つため，適切に管理することで蓄積を防ぐ必要がある．技術的負債の蓄積を防ぐための手段として，コードレビューによりコードのクオリティを担保することが挙げられる．このうち，Modern Code Review（MCR）は形式的なコードレビューとは異なる簡易的なコードレビューを指し，スケジュールの合わせやすさやレビューにかかるコストの低さといった利点を持つため近年よく用いられている．しかし，現状ではSATDとMCRとの関連を調べた研究はほとんどない．そこで，MCRにおけるSATDの性質を知るため，新たな知見の獲得を試みた．結果，1) 変更にSATDを含むレビューはそうでないレビューと比べて不採録率が高く，修正回数が多くなりやすい傾向があること，2)MCRにおいて，SATDの削除は負債の対処以外にも関数やファイルの削除により発生することが多いこと，3)SATDの追加は作業途中に後で処理するものとして発生しやすいことがわかった．

1. はじめに

プロジェクトを開発する中で，開発者は短期的な目的を達成するために，設計上最適ではない次善策を用いることがある．例えば，即座には解決できない不具合に遭遇して回避策を取ったり，素早く機能を実装する必要に迫られて最適でない実装方法を選択する，といったものである．このような開発上における設計状態の理想状態からの乖離を示す現象は技術的負債と呼ばれる [1]．

負債という単語が使われる通り，技術的負債には対処までの時間が長くなるほど処理にかかるコストが増加するという傾向を持つ [2] ため，技術的負債は適切に認識，管理および処理される必要がある．過去に技術的負債を検出する方法として，コードの不吉な臭いやコーディングスタイル違反といった指標が用いられてきた [3][4]．その中でもコメントを用いて技術的負債を探る方法として，Self-Admitted Technical Debt（SATD）を用いる方法が挙げられる．

SATDは技術的負債の中でもコメントを用いて自ら言及が行われており，開発者自身により認識されている技術的負債のことを指す [5]．コメントを用いてソースコード上の不備を示す開発者は多く，Vassallaoら [6] が金融機関の開発者に対して行った調査では回答者中88%が不適切な実装に対し，可能な場合に修正するために注釈をつけているということが示されている．このため，コメントを基としたSATDについての研究は技術的負債の処理に有用であると考えられる．加えて，SATDコメントは修正が発生しやすいファイルやメソッドの発見に有用であるという知見が得られており [7]，SATDコメントを用いることが負債の発見に役立つと考えられる．

技術的負債の検出後は，負債の増加を防ぐために適切な管理が必要となる．検出した技術的負債の管理を行う手段としてコードレビューが考えられる．コードレビューはソフトウェア開発プロセスにおけるソフトウェアの品質の担保活動である．コードの変更を反映する前に，開発者間で変更内容の確認や批評を行うことで早期に欠陥を見つけ出し，修正を行うことにより品質を担保する．従来は高度に構造化された形式的なコードレビューが用いられてきたが，形式的なコードレビューでは時間が多くかかる．加えて，人数に比例してスケジュールを合わせにくいため，分散した環境下でのソフトウェア開発では扱いにくいという欠点が存在する [8]．

そのため近年では，より軽量で略式的な，コードレビュー

[1] 九州大学
Kyushu University
[a)] nishikawa@posl.ait.kyushu-u.ac.jp
[b)] kamei@ait.kyushu-u.ac.jp
[c)] sato@ait.kyushu-u.ac.jp
[d)] ubayashi@ait.kyushu-u.ac.jp

ツールを利用した手法として，モダンコードレビュー（Modern Code Review, 以下 MCR）が用いられている [9]．MCR では欠陥の発見以外にも，知識の伝達や問題への代わりの解決法の作成などといった利点を持つ．その性質上，オープンソースソフトウェア（OSS）のような遠隔環境で開発が行われるプロジェクトにおいてソフトウェアの品質を向上する助けとなっている．

しかし，MCR とコード品質に関する研究は多いものの，MCR の環境下で SATD がどのような性質を持つのか，どのように対処されるかについて着目した研究はほとんどない．そこで本研究では，MCR を採用しているプロジェクトに着目して SATD の調査を行うことで，MCR 環境下での SATD の性質について新たな知見の獲得を試みた．

Openstack と Qt の 2 プロジェクトのコードレビューデータから SATD を抽出し，(RQ1) 全コードレビューのうち，変更に SATD を含むコードレビューは何件存在するのか，(RQ2) 変更に SATD が含まれるコードレビューは，その他のコードレビューと比べて，不採録率が高いのか，また，修正回数が多いのか，(RQ3) レビュー途中で削除，および追加される SATD はどの程度存在するのか，についての調査を行った．

以降，第 2 章では関連研究について述べる．第 3 章では本研究の実験設計，使用したデータセット，調査課題と各調査課題のアプローチ，そして得られた結果を述べる．第 4 章では得られた結果のまとめと意義について述べる．最後に第 5 章では本研究のまとめと今後の展開を述べる．

2. 関連研究

Self-Admitted Technical Debt. Potdar ら [5] は急な修正や一時的な修正による誤りの影響についての研究が過去に十分行われていなかったという観点から，Self-Admitted Technical Debt を提唱し，約 10 万のコメントを分析して SATD コメントのパターンを 62 種類に分類した．加えて，4 つのプロジェクトに対して調査を行い，2.4〜31.0%のファイルに SATD が含まれていたこと，経験の多い開発者ほど SATD の追加を行うことが多いこと，リリースまでの時間やコードの複雑さには SATD に対して強い相関がなく，リリース後にも SATD の削除が行われていることを示した．

Maldonado ら [10] は，SATD がどのような種類の技術的負債を発見することに役立つかが明らかになっていなかったという理由から，5 つのプロジェクトを対象に調査を行った．SATD を設計，欠陥，文書化，要求，テストといった 5 つの種類に分け，中でも設計負債が最も多く，次点で要求負債が多いことを示した．

Modern Code Review. McIntosh ら [11] は，過去に MCR の特性とソフトウェアの品質の関係を定量的に調査している研究がほとんどなかったことから，コードの欠陥とコードレビューの関係に着目し，研究を行った．レビューの参加率やレビューが行われたコードの範囲に対するリリース後の欠陥の関係，レビュー参加者とリリース後の欠陥の関係について調べ，結果，変更箇所のうちレビューされた割合とソフトウェア品質との間に負の相関があること，欠陥が出やすい一部のコンポーネントではレビューされた変更の割合が大きいこと，コードレビューへの参加者の割合の低さがソフトウェア品質に悪影響を与えること，および議論が行われていないレビューが欠陥に繋がりやすいことを示した．

また，Patanamon ら [12] は，MCR へのレビュー参加に影響する要因を解明するため，レビュアーが参加しにくくなりやすいパッチ特性，議論がなされにくくなるパッチ特性，フィードバックが遅くなるパッチ特性を調べた．その結果，前回の変更からの日数がレビューへの参加率に影響を与えること，パッチの説明が長いと議論されやすくなること，新しい機能の導入を目的としたパッチはフィードバックが遅くなることを示した．

これらの研究のように，実際に発生した欠陥やパッチへの直接的な影響を調べた研究は複数存在するが，技術的負債のような，プロジェクトへの間接的な阻害要因へ焦点を置いた調査は行われていない．このため，本研究では MCR と SATD との関係性に着目して調査を実施する．

3. 実験設計

本章では，本研究における調査課題（以下 RQ），および使用したデータセットについて説明する．本研究では，以下の調査を行った．

RQ1：全コードレビューのうち，変更に SATD を含むコードレビューは何件存在するのか？

RQ2：変更に SATD が含まれるコードレビューは，その他のコードレビューと比べて，不採録率が高いのか？また，修正回数が多いのか？

RQ3：レビュー途中で削除，および追加される SATD はどの程度存在するのか？

3.1 (RQ1) 全コードレビューのうち，変更に SATD を含むコードレビューは何件存在するのか？

動機 :SATD が関わるレビューがほとんど存在しない場合，以降の RQ からコードレビューと SATD の関係を調べても役立つ知見とならず，研究価値としては限定的なものとなる．そのため，まず初めに変更に SATD がどの程度関わっているのかを調べる必要がある．

アプローチ :データの取得プロセスを図 1 に示す．最初に研究に使用するレビューデータを収集する．使用するデータについては，レビューが行われる割合が高く，活発にプロジェクトが発展していることから，McIntosh ら [13] の先行研究を参考に Openstack と Qt を選択した．レビュー

データの取得には Gerrit の API を使用した．データの取得後は，パッチ毎にコメントの差分を抽出し，その中に SATD コメントが存在するかを調べる．レビュー開始から最終的にマージされるか棄却，または放置されるまでを 1 つのレビューとして扱い，その中の全パッチセットの変更箇所から SATD コメントを検出する．

SATD コメントの検出には SATD Detector[14] を用いた．SATD Detector は機械学習を用いた，SATD コメントを判別する Java ライブラリである．学習データとして ArgoUML, Columba, Jmeter, JFreeChart, Hibernate, Jedit, JRuby, SQuirrel の 8 プロジェクトから単語を抽出している．さらに，ベクトル空間モデルを用いた特徴選択により単語から有用な特徴を抽出し，それらの単語を用いて，サブ分類器である複数のナイーブベイズ分類器をプロジェクトごとに学習させることでモデルを構築している．入力にはコメントとなる文字列を受け付けており，コメントを受け取ると特徴を抽出し，複数のサブ分類器に入力し，サブ分類器ごとに入力が SATD コメントであるか否かを判定し，その多数決により最終的に SATD コメントであるか否かを決定する．

SATD Detector の判定で 1 つでも SATD コメントが含まれていれば変更に SATD を含むレビューであるとみなす．結果の評価については Herzig ら [15] の研究を参考に 1%を基準点とし，SATD を含むレビューの割合が 1%以上であるならば SATD の影響を無視できないものと考える．

結果 :結果を表 1 に示す．Openstack と Qt の両方において，10%前後のレビューが変更に SATD を含んでいることがわかった．このことから，少なからず SATD を含むレビューが存在し，SATD の影響を無視できないものと考えられる．また，Openstack と Qt で変更に SATD を含む割合に差があるが，この理由についてはレビュー品質の違いが挙げられる．Potdar ら [5] の研究では経験の多い開発者ほど SATD を追加しやすい傾向があることが示されている．Openstack に比べ Qt では小規模のレビューが多く，レビュー後半まで議論が行われない，セルフレビューに近いレビューが多かった．このため，よりレビュー品質の高い Openstack において変更に SATD を含むレビューの割合が多くなったものと考えられる．

SATD が関わるレビューの例を図 2 と図 3 に示す．それぞれ Openstack での SATD の削除，追加の例となる．図 2 は SATD の削除が発生している例である．赤く塗りつぶされた部分が後のパッチにて削除された箇所となる．変更前のコードでの SATD コメントにあるように，このコードにはメジャーバージョンの変化に合わせて返り値を変更しなければならないという SATD が存在するが，変更後ではコードが削除されることにより SATD が消滅している．

図 3 は SATD の追加が発生している例である．緑で塗りつぶされた部分がレビュー中のパッチにより追加された箇所となる．変更前と比べてコードが 1 行追加されており，追加したコードへの必要性を説明しなければならない旨がその上に TODO として記されている．

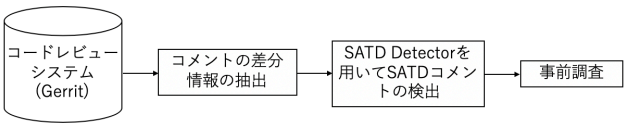

図 1　RQ1 のデータ取得プロセス

表 1　RQ1 結果

Project	# Reviews	# With SATD	% With SATD
Openstack	79,644	10,730	13.5
Qt	86,274	7,913	9.2

```
127  def vol_usage_update(self, context, vol_id, rd_req, rd_bytes, wr_req,
128                       wr_bytes, instance, last_refreshed=None,
129                       update_totals=False):
130      return self._manager.vol_usage_update(context, vol_id,
131                                            rd_req, rd_bytes,
132                                            wr_req, wr_bytes,
133                                            instance, last_refreshed,
134                                            update_totals)
135
136  def service_get_by_compute_host(self, context, host):
137      result = self._manager.service_get_all_by(context, 'compute', host,
138              binary=None)
139      # FIXME(comstud): A major revision bump to 2.0 should return a
140      # single entry, so we should just return 'result' at that point.
141      return result[0]
142
143  def compute_node_create(self, context, values):
144      return self._manager.compute_node_create(context, values)
145
```

図 2　SATD が削除される例

```
146
147      return webob.Response(request=request, status=status,
148                            content_type=content_type,
149                            body=body)
150  #TODO(blogan): explain the need for this
151  setattr(resource, 'controller', controller)
152  return resource
153
```

図 3　SATD が追加される例

Openstack では 13.5%，Qt では 9.2%のレビューが変更に SATD を含んでいた．

3.2　(RQ2) 変更に SATD が含まれるコードレビューは，その他のコードレビューと比べて，不採録率が高いのか？また，修正回数が多いのか？

動機 :SATD の多くには技術的負債の利子が存在するため，放置していると SATD が増幅する傾向がある [2]．このため，直感的には SATD の削除や追加には注意が払われ，SATD が変更に含まれると変更についての議論や指摘がなされることで修正回数が多くなる，もしくは採録されにくくなる傾向があると予想されるが，実際がどうであるかは定かではない．以上のことを知ることにより，SATD を変更に含むレビューが他のレビューと比べてどの程度コストが多くかかるか否かを見積もることができる．

アプローチ :RQ1 で使用した全レビューを，SATD を含むレビューと SATD を含まないレビューの 2 群に分ける．こ

れらそれぞれについて Gerrit API で取得したデータを適用する．採録・不採録については API 上でステータスが MERGED と記されているもののみを採録として扱い，その他を不採録とし，不採録であるレビューの割合を不採録率とする．修正回数については各レビューの総パッチセット数を調べ，中央値を求めることで比較する．

結果 :結果を表 2 に示す．不採録率については，Openstack では変更に SATD を含むレビューでは 31.1%，変更に SATD を含まないレビューの不採録率が 25.9%となった．Qt では SATD を含むレビューの不採録率が 26.0%，変更に SATD を含まないレビューの不採録率が 18.5%，となった．不採録率の差をより詳細に評価するため，それぞれのプロジェクトで比率の差の検定を行った．結果を表 3 に示す．Openstack と Qt の両方において有意差が見られ (p<0.05)，変更に SATD を含むレビューが SATD を含まないレビューに比べて不採録になる割合が高いことが示された．

パッチセットについては，Openstack では変更に SATD を含むレビューのパッチセット数の中央値が 4.00 であるのに対し，変更に SATD を含まないレビューのパッチセット数の中央値が 2.00 となった．Qt でも似たような結果が得られ，変更に SATD を含むレビューのパッチセット数の中央値が 3.00 であるのに対し，変更に SATD を含まないレビューのパッチセット数の中央値は 2.00 となった．これらの修正回数の差をより詳細に評価するため，それぞれのプロジェクトでマン・ホイットニーの U 検定を行った．結果を表 4 に示す．Openstack と Qt の両方において有意差が見られ (p<0.05)，変更に SATD を含むレビューが SATD を含まないレビューに比べ，修正回数が多いという結果となった．

追加分析 :上述の調査について，変更に SATD を含むパッチはそれ以外のパッチに比べパッチサイズが大きくなる傾向が存在するため，SATD ではなくパッチサイズが直接不採録率や修正回数に影響を与えている可能性がある．この可能性を排除するため，追加で SATD の有無とパッチサイズに対する不採録率，修正回数の関係をロジスティック回帰分析，および重回帰分析にて調べた．回帰分析の結果を表 5 と表 6 に示す．それぞれ，説明変数には変更に SATD を含むか否かを示したダミー変数と，レビュー開始前のコードに対する最終パッチの差分行数の対数を取ったものを使用している．不採録率と修正回数の両方において，Openstack と Qt 共に SATD の回帰係数は 95%信頼区間内で正の値を取っており，変更に SATD が含まれることでレビューが採録されにくくなり，修正回数が多くなることが示されている．

表 2 RQ2 結果

Project	SATD を含むか否か	# Reviews	不採録率 (%)	パッチセット数 (中央値)
Openstack	含む	10,730	31.1	4.00
	含まない	68,914	25.9	2.00
Qt	含む	7,913	26.0	3.00
	含まない	78,361	18.5	2.00

表 3 不採録率に対する比率の差の検定 結果

Project	カイ二乗値	p 値	95%信頼区間
Openstack	126.95	1.905e-29	[0.043, 0.060]
Qt	260.23	1.527e-58	[0.066, 0.084]

表 4 修正回数に対するマン・ホイットニーの U 検定 結果

Project	Z 統計量	p 値
Openstack	51.217	4.726e-764
Qt	42.313	3.088e-391

表 5 不採録率を目的変数としたロジスティック回帰分析 結果

Project	説明変数	回帰係数	95%信頼区間	オッズ比
Openstack	SATD	0.2342	[0.187, 0.281]	1.2639
	Patch size	0.0111	[0.002, 0.020]	1.0112
Qt	SATD	0.3061	[0.249, 0.363]	1.3581
	Patch size	0.0586	[0.050, 0.067]	1.0603

表 6 修正回数を目的変数とした重回帰分析 結果

Project	説明変数	回帰係数	95%信頼区間
Openstack	SATD	3.0564	[2.906, 3.207]
	Patch size	1.0360	[1.008, 1.064]
Qt	SATD	1.6085	[1.529, 1.689]
	Patch size	0.4225	[0.411, 0.434]

Openstack と Qt の両方において，変更に SATD が含まれるコードレビューはそのほかのコードレビューと比べて不採録率が高く，修正回数も多かった．

3.3 (RQ3) コードレビューの途中で削除・追加される SATD の数・割合はどの程度か？どのような理由で削除・追加されるのか？

動機 :コードレビューはプロジェクトの品質を保つために行われるものである．それが SATD に対しても効果を持つものであるのか否かを知りたい．もし SATD がコードレビューに関係なく削除・追加されるものである場合，コードレビューが SATD に対して効果を発揮しにくいと言える．その場合，開発者が SATD の管理不足によるプロジェクトの品質の悪化に対して注意を払う必要があることを示唆できる．

アプローチ :RQ1 で変更に SATD を含むとされたレビューのそれぞれに対し，どのパッチで削除・追加されたのかを調べる．そのレビューでのすべての SATD コメントが最

表 7　RQ3 コードレビューの途中で削除された SATD の数・割合

Project	# SATD の削除を含むレビュー	# レビュー中に削除	# レビュー依頼時に削除	% レビュー中に削除
Openstack	3,868	989	2,879	25.6
Qt	3,163	595	2,568	18.8

表 8　RQ3 コードレビューの途中で追加された SATD の数・割合

Project	# SATD の追加を含むレビュー	# レビュー中に追加	# レビュー依頼時に追加	% レビュー中に追加
Openstack	7,209	3,518	3,691	48.8
Qt	4,926	1,924	3,002	39.1

表 9　RQ3 レビュー途中での SATD 削除の理由について

Reason	Openstack	Qt	Total
Addressed	35	23	58
Deleted	26	30	56
Conflict	0	9	9
Ignored	5	3	8
Improvement	3	3	6
Others	1	2	3
Total	70	70	140

初のパッチで削除・追加されていた場合は，レビュー依頼時に削除・追加されたレビューとし，1 つ以上の SATD コメントが 2 番目以降のパッチで削除・追加されていた場合は，レビュー途中で SATD が削除・追加されたレビューとする．

その後，レビュー途中で削除・追加された SATD をランダムに選択し，SATD コメント本文，コードの変化内容，コミットメッセージ，レビューコメントを用いた目視でその理由を調査する．この際，判断材料が十分でないなどで理由を判断できなかった 49 件の SATD は結果から除外した．なお，結果は削除と追加で別々にとり，セルフレビューは他者によるレビューが行われない性質上，本来のコードレビューとは異なる目的で利用されていることが考えられるため結果から除外する．以上の除外処理を経て残った，各プロジェクト 70 件の SATD について目視調査を行った．

結果：コードレビューの途中で削除された SATD の数・割合についての結果を表 7 に示す．レビュー途中で削除が行われる SATD の割合は 20%前後で，多くの SATD がレビュー依頼時の時点で削除が行われていることが分かった．

コードレビューの途中で追加された SATD の数・割合についての結果を表 8 に示す．レビュー途中で追加が行われる SATD の割合は約 40%〜50%で，削除に比べてコードレビュー中に追加されることが多いということが分かった．

> コードレビューの途中で削除された SATD の割合は Openstack では 25.6%，Qt では 18.8%であった．また，コードレビューの途中で追加された SATD の割合は Openstack では 48.8%，Qt では 39.1%であった．

削除の理由について：目視調査にてレビュー途中で SATD が削除された理由を調べたところ，SATD が対処された (Addressed)，ファイルや機能の削除に巻き込まれた (Deleted)，マージコンフリクト解消の際に削除された (Conflict)，SATD が無視された (Ignored)，リファクタリング等のコードクオリティ向上の一環で削除された (Improvement)，といった大きく 5 つの理由に分けられた．表 9 に Openstack と

スニペット 1　SATD の削除例 1

```
1 -   # TODO(geguileo): In O set added_to_cluster to True
2     # We don't want to include resources in the cluster
            during the
3     # start while we are still doing the rolling upgrade.
4 -   self.added_to_cluster = not self.is_upgrading_to_n
5 +   self.added_to_cluster = True
```

スニペット 2　SATD の削除例 2

```
1 - def get_ports(self, context, filters=None, fields=None):
2 -   with context.session.begin(subtransactions=True):
3 -       ports = super(NECPluginV2, self).get_ports(context,
          filters, fields)
4 -       # TODO(amotoki) filter by security group
5 -       for port in ports:
6 -           self._extend_port_dict_binding(context, port)
7 -       return [self._fields(port, fields) for port in ports]
```

スニペット 3　SATD の削除例 3(コード)

```
1 -   <<<<<<< HEAD
2 -   ||||||| merged common ancestors
3     Q_LOGGING_CATEGORY(qLcEglfsKmsDebug, "qt.qpa.eglfs.kms")
4
5 -   =======
6 -   // Use a name different from qLcEglfsEglKmsDebug to avoid
          duplicate symbols in
7 -   // static builds. Starting from Qt 5.7 this will be
          solved by the common kms
8 -   // support library, but in the meantime just work it
          around.
9 -   Q_LOGGING_CATEGORY(qLcEglfsEglDevDebug, "qt.qpa.eglfs.kms
          ")
10 -
11 -  >>>>>>> origin/5.6
```

スニペット 4　SATD の削除例 3(コミットメッセージ)

```
1 Merge remote-tracking branch 'origin/5.6' into 5.7
2
3 Conflicts:
4     src/angle/src/libGLESv2/libGLESv2.pro
5     src/plugins/platforms/eglfs/deviceintegration/
            eglfs_kms_egldevice/qeglfskmsegldeviceintegration
            .cpp
6
7 Change-Id: If8da4cfe8f57fea9f78e7239f378a6302c01674e
```

スニペット 5　SATD の削除例 4

```
1 -   # TODO(twilson) DEFAULT.ovs_vsctl_timeout should be OVS.
          vsctl_timeout
2     cfg.CONF.import_opt('ovs_vsctl_timeout', 'neutron.agent.
            common.ovs_lib')
```

スニペット 6　SATD の削除例 5(コード)

```
- def test_update_server_adminPass_ignored(self):
-     ...
-     # FIXME (comstud)
-     #       self.stubs.Set(db, 'instance_get',
-     #                      return_server_with_attributes(name='
      server_test'))
-     ...
```

スニペット 7　SATD の削除例 5(コミットメッセージ)

```
Refactor test_update_* from test_servers.py in current API
    tests and v3.

Cleanup from copy-paste code.
Move all test_update to separate class.
```

Qt でのそれぞれの内訳を示す．

Openstack と Qt の両方において，SATD が対処された，ファイルや機能の削除に巻き込まれたといった理由が主な削除の理由として挙げられ，SATD の削除が必ずしも問題の解決を示しているとは限らないことがわかった．以下にそれぞれの詳細を示す．

- **Addressed:** 負債内容が直接対処されたことが理由で削除された SATD が該当する．
  Openstack では 35 件，Qt では 23 件の合計 58 件がこの理由に該当した．そのうちの一例をスニペット 1 に示す．スニペット 1 では，O(Openstack のバージョン Ocata を指す) になったら変数 self.added_to_cluster を True にするという TODO が SATD コメントとして存在し，変更によりその SATD コメントが削除されると同時に変数 self.added_to_cluster が True になっている．この変更は SATD に直接対処するものであるため，Addressed として分類される．
- **Deleted:** SATD を含んでいた関数やファイルが必要なくなったこと等が理由で，関数やファイルの削除に巻き込まれる形で削除された SATD が該当する．
  Openstack では 26 件，Qt では 30 件の合計 56 件がこの理由に該当した．そのうちの一例をスニペット 2 に示す．スニペット 2 では，ポートに対するフィルタリングが行われていないことが SATD コメントにより示されている．しかし，関数ごと削除される形で SATD が消滅している．この場合，Deleted として分類される．
- **Conflict:** マージコンフリクトを解消する際に削除された SATD が該当する．
  Qt でのみ，9 件がこの理由に該当した．そのうちの一例をスニペット 3，スニペット 4 に示す．スニペット 3 が SATD の削除があったコード，スニペット 4 が SATD の削除があったパッチのコミットメッセージである．スニペット 3 では qLcEglfsEglDevDebug という名前についての不適切さが SATD コメントで示されており，マージコンフリクトの解消により削除されている．また，スニペット 4（コミットメッセージ）ではマージコンフリクトの解消を目的としたコミットである旨が記されている．以上の場合，マージコンフリクトの解消の際に削除された SATD として，Conflict に分類される．
- **Ignored:** 負債内容が対処されていないにも関わらず SATD コメントのみが削除され，なおかつその理由がレビューコメントやコード変更履歴から推測できない SATD が該当する．
  Openstack では 5 件，Qt では 3 件の合計 8 件がこの理由に該当した．そのうちの一例をスニペット 5 に示す．スニペット 5 では命令文の 1 つ目の引数を変更すべきであるという旨の TODO が SATD コメントとして示されているが，実際に変更されることは無く SATD コメントのみが削除されている．以上のようにコードへの変更がなく SATD コメントのみが削除されていた場合，Ignored に分類される．
- **Improvement:** リファクタリング等，コードが改善される中で削除された SATD が該当する．なお，Improvement と前述の Deleted が両方とも該当する場合は，こちらを優先する．
  Openstack では 3 件，Qt では 3 件の合計 6 件がこの理由に該当した．そのうちの一例をスニペット 6，スニペット 7 に示す．スニペット 6 が SATD の削除があったコード，スニペット 7 が SATD の削除があったパッチのコミットメッセージである．スニペット 6 では，未修正のコードが存在することが FIXME という SATD コメントで示されているが，関数ごと削除される形で SATD が消滅している．ここまでは先ほどの Deleted と同じであるが，加えてスニペット 7（コミットメッセージ）にて，test_update_*という名前の関数を対象にリファクタリングを行った旨が記されている．以上の場合，リファクタリングの一環で削除された SATD として，Improvement に分類される．

> SATD が削除された理由は SATD の対処，ファイルや機能の削除に巻き込まれた事による削除，マージコンフリクト解消の一環，SATD の無視，リファクタリング等のコードクオリティ向上の一環，といったものに分けられた．特に，SATD の対処 (41.4%) や，ファイルや機能の削除に巻き込まれた事による削除 (40.0%) が多く存在した．

追加の理由について：目視調査にてレビュー途中で SATD が追加された理由を調べたところ，作業途中である (WIP)，対処に他のタスクの進行等を待つ必要がある (Waiting)，

表 10　RQ3 レビュー途中での SATD 追加の理由について

Reason	Openstack	Qt	Total
WIP	27	28	55
Waiting	15	8	23
Doubt	8	13	21
Discussed	17	3	20
Unsolvable	1	5	6
Others	2	13	15
Total	70	70	140

スニペット 8　SATD の追加例 1(コード)

```
1 +       # TODO(gibi): We need to make sure that the
          requested_resources field
2 +       # is re calculated based on neutron ports.
3 +       self.request_spec.requested_resources = []
```

スニペット 9　SATD の追加例 1(コミットメッセージ)

```
1 Transfer port.resource_request to the scheduler
2 ...
3 This patch only handles the happy path of a server create
          request. But
4 it adds couple of TODOs to places where the server move
          operations
5 related code pathes need to be implemented. That
          implementation will be
6 part of a subsequent patches.
```

スニペット 10　SATD の追加例 2

```
1 +    # The required version is a bit tricky but we know that
          we at least
2 +    # need 1.0 for Newton computes. This minimum might
          change in the
3 +    # future.
4 +     needs_version = 1.0
```

スニペット 11　SATD の追加例 3

```
1 +       #TODO(justinsb): Is this always 1? Does it matter?
2 +       iscsi_portal_interface = '1'
```

スニペット 12　SATD の追加例 4(コード)

```
1 +   // TODO: Merge with qt_xcb_imageFormatForVisual in
          qxcbimage.cpp
2     static inline QImage::Format imageFormatForVisual(int
            depth, quint32 red_mask, quint32 blue_mask,  bool
             &rgbSwap, bool endianSwap)
3     {
4         ...
```

コード内容に疑問がある (Doubt)，他者に指摘を受けた (Discussed)，自力で対処が行えない (Unsolvable)，といった大きく 5 つの理由に分けられた．

表 10 に Openstack と Qt でのそれぞれの内訳を示す．Openstack と Qt の両方において，SATD が追加された最も主な理由は作業途中で後に処理するものとして SATD

スニペット 13　SATD の追加例 4(レビューコメント)

```
1 Gatis Paeglis  Jul 03, 2017 20:56
2 Patch Set 2:
3 I still think this needs some code unification. Have you seen
           the byte swapping code from http://code.qt.io/cgit/
          qt/qtbase.git/tree/src/plugins/platforms/xcb/
          qxcbimage.cpp#n110 ?
4
5 Allan Sandfeld Jensen(author) Jul 03, 2017 21:38
6 Patch Set 2:
7 No, I hadn't seen that. There seems to be some duplication
          between windows and images, but currently they are
          different code paths.
```

スニペット 14　SATD の追加例 5

```
1 +   except UnicodeEncodeError:
2 +       self.logger.warn("xml field %s value %s \
3 +       can't be utf-8 decoded." % (field, orig))
4 +       # TODO - find out why this happens in functests
```

を追加したというものであった．以下にそれぞれの詳細を示す．

- **WIP:** レビュー終了時までに解決予定であると考えられる，単純に作業途中であることが理由で追加された SATD が該当する．
  Openstack では 27 件，Qt では 28 件の合計 55 件がこの理由に該当した．そのうちの一例をスニペット 8，スニペット 9 に示す．スニペット 8 が SATD の追加があったコード，スニペット 9 が SATD の追加があったパッチのコミットメッセージである．スニペット 8 ではコードの追加と同時に，requested_resources フィールドを確認する必要がある旨が SATD コメントで示されている．加えて，スニペット 9（コミットメッセージ）では，一部の TODO を後のパッチで解決する旨が示されている．以上のように，実装途中で後回しにされていることがわかる場合，WIP に分類される．
- **Waiting:** 負債内容の解決のためには別タスクの問題の解決やすぐには行えない修正，将来的なアップデート等を待つ必要があることが分かっており，その時点で解決できないことが理由で追加された SATD が該当する．
  Openstack では 15 件，Qt では 8 件の合計 23 件がこの理由に該当した．そのうちの一例をスニペット 10 にを示す．スニペット 10 では，動作に必要なバージョンの値が将来的に変更される可能性があることを SATD コメントで示している．以上のように，現時点ではなく将来的な変更に応じて変更を行う必要がある場合，Waiting に分類される．
- **Doubt:** 開発者がコードの正しさやクオリティに疑念を持ったことにより追加された SATD が該当する．
  Openstack では 8 件，Qt では 13 件の合計 21 件がこの

理由に該当した．そのうちの一例をスニペット 11 に示す．スニペット 11 では，変数の値に対して TODO を用いて疑問を呈している．このように，コードの正しさに対して疑問があることが示唆されている場合，Doubt に分類される．

- **Discussed:** TODO を入れる指示や提案を受けた，コードに関する議論の結果 SATD として残すことになった等，他者の干渉を受けて追加された SATD が該当する．
  Openstack では 17 件，Qt では 3 件の合計 30 件がこの理由に該当した．そのうちの一例をスニペット 12，スニペット 13 に示す．スニペット 12 が SATD の追加があったコード，スニペット 13 が SATD が追加される直前のパッチでのレビューコメントである．スニペット 13（レビューコメント）では機能の重複が指摘されており，その後スニペット 12 においてその旨が TODO として追加されている．以上のように，他者の指摘や議論を経て追加されている SATD の場合，Discussed に分類される．
- **Unsolvable:** 原因がわからない等，開発者が自力で解決できないことが理由で追加された SATD が該当する．
  Openstack では 1 件，Qt では 5 件の合計 6 件がこの理由に該当した．そのうちの一例をスニペット 14 に示す．スニペット 14 では，UnicodeEncodeError が発生する理由を調べる必要がある旨を示している．以上のように，原因を究明できていない問題が存在することを示唆されている場合，Unsolvable に分類される．

SATD が追加された理由は作業途中である，コード内容への疑念，他タスクの進行待ち，自力で対処が行えない，他者からの指摘といった理由に分けられた．特に，作業途中で後で実装を行うことを示すための SATD の追加が多かった (39.3%)．

4. 結果のまとめと意義

本章では，本研究で得られた結果について開発者，研究者のそれぞれに対してどのように役立つかを説明する．

開発者への貢献 : 開発者への貢献となる結果として，SATD は新しい機能の作成途中で追加されやすく，SATD の削除時にはレビュー依頼時点で削除を行いやすいこと，負債に直接対処すること以外を目的とした削除が半数ほど存在することが挙げられる．このことから，もし SATD が偶発的かつ不適切に削除された場合，技術的負債としての管理が難しくなってしまうため，レビュアーは SATD の削除に対して注意を払う必要があると言える．

研究者への貢献 : 研究者への貢献となる結果として，MCR においては変更に SATD が含まれていると不採録率が高く，修正回数が多いこと，SATD の対処やファイル削除に巻き込まれる形の削除が多いこと，作業途中での SATD の追加が多いこと，レビュー内での討論の結果追加された SATD が存在することが挙げられる．特に，不採録率および修正回数からは SATD が含まれることによりレビューが慎重になる傾向があること，討論により追加される SATD が存在することから MCR が SATD の把握に貢献があることが言える．以上から，MCR 環境下においては SATD に対して管理がより厳密になっている可能性が考えられる．

5. 妥当性への脅威

本章では，妥当性への脅威について議論する．

5.1 内的妥当性

内的妥当性では，調査結果へ影響を与えた可能性がある要因について議論する．

一つ目に，SATD の検出精度が挙げられる．今回使用した SATD Detector は機械学習を用いて作られており，過去に Huang ら [16] が ArgoUML, Columba, Jmeter, JFreeChart, Hibernate, Jedit, JRuby, SQuirrel の 8 プロジェクトのうち 7 プロジェクトを学習データに，残りの 1 プロジェクトをテストデータに用いて行った精度評価では，F 値は平均 0.737，precision は平均 0.756，recall は平均 0.733 となっている．このため，SATD の検出の精度がツールの精度に依存する．

二つ目に，SATD の削除および追加の理由についての分類が挙げられる．理由の分類は目視により行われている．そのため，人力による個人的なバイアスがかかっている可能性が考えられる．

5.2 外的妥当性

外的妥当性では調査結果の一般化において影響を与える事柄について議論する．

外的妥当性への脅威となりうる要因として，プロジェクトの数や範囲が挙げられる．今回は Openstack および Qt といった二つのプロジェクトを対象に調査を行っている．これらはいずれも Gerrit を用いた OSS である．結果の一般化のためには，調査対象のプロジェクト数を増やし，Gerrit 以外の MCR システムを使用しているプロジェクトでも同様に調査を行う必要がある．

6. おわりに

本研究では，MCR 環境下での SATD が持つ性質について，変更に SATD が含まれることにより採録率や修正回数が変化するのか，レビュー途中での SATD の削除や追加がどの程度存在するのかを調査した．今後の課題としては，データセットの拡大により結果の幅を広げ一般化すること

や，MCR 環境下で発生する SATD とそれ以外の SATD を比較して MCR による SATD への影響をより詳細に調査することが挙げられる．

謝辞

本研究は，JSPS 科研費 JP18H03222，および JSPS・国際共同研究事業の助成を受けたものである．

参考文献

[1] Cunningham, W.: The WyCash portfolio management system, *OOPS Messenger*, Vol. 4, No. 2, pp. 29–30 (1993).

[2] Kamei, Y., da S. Maldonado, E., Shihab, E. and Ubayashi, N.: Using Analytics to Quantify Interest of Self-Admitted Technical Debt, *Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2016) and 1st International Workshop on Technical Debt Analytics (TDA 2016) co-located with the 23rd Asia-Pacific Software Engineering Conference (APSEC 2016)*, pp. 68–71 (2016).

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

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

[5] Potdar, A. and Shihab, E.: An Exploratory Study on Self-Admitted Technical Debt, *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, ICSME 2014*, pp. 91–100 (2014).

[6] Vassallo, C., Zampetti, F., Romano, D., Beller, M., Panichella, A., Penta, M. D. and Zaidman, A.: Continuous Delivery Practices in a Large Financial Organization, *Proceedings of IEEE International Conference on Software Maintenance and Evolution, ICSME 2016*, pp. 519–528 (2016).

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

[8] Votta, L. G.: Does Every Inspection Need a Meeting?, *SIGSOFT '93, Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering, 1993*, pp. 107–114 (1993).

[9] Bacchelli, A. and Bird, C.: Expectations, outcomes, and challenges of modern code review, *35th International Conference on Software Engineering, ICSE 2013*, pp. 712–721 (2013).

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

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

[12] Thongtanunam, P., McIntosh, S., Hassan, A. E. and Iida, H.: Review participation in modern code review - An empirical study of the android, Qt, and OpenStack projects, *Empirical Software Engineering*, Vol. 22, No. 2, pp. 768–817 (2017).

[13] McIntosh, S. and Kamei, Y.: Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction, *IEEE Trans. Software Eng.*, Vol. 44, No. 5, pp. 412–428 (2018).

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

[15] Herzig, K. and Zeller, A.: The impact of tangled code changes, *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR 2013*, pp. 121–130 (2013).

[16] Huang, Q., Shihab, E., Xia, X., Lo, D. and Li, S.: Identifying self-admitted technical debt in open source projects using text mining, *Empirical Software Engineering*, Vol. 23, No. 1, pp. 418–451 (2018).