社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

コンテナ仮想化技術における Self-Admitted Technical Debt の調査

東 英明† 柗本 真佑† 亀井 靖高†† 楠本 真二†

† 大阪大学大学院情報科学研究科
†† 九州大学大学院システム情報科学研究院
E-mail: †{h-azuma,shinsuke,kusumoto}@ist.osaka-u.ac.jp, ††kamei@ait.kyushu-u.ac.jp

あらまし ソフトウェア開発において，開発者が意図的に混入させた場当たり的な解決策を，Self-Admitted Technical Debt（SATD）と呼ぶ．SATD の存在は場当たり的な実装の波及に繋がるため，可能な限り早く除去する必要がある．他方，近年サーバ等のインフラを支える技術として，コンテナ仮想化と呼ばれる仮想化技術が注目を浴びている．この技術において，コンテナの構築方法が記述されたファイルは手続き的な命令の集合であり，一種のソースコードであると見なせる．そのため，既存の SATD 研究で調査対象とされている一般的なプログラミング言語と同様に SATD が多数存在すると考えられる．本研究の目的は，コンテナ仮想化技術における SATD の体系化，及びその知見の共有である．目的達成のために，コンテナ仮想化技術のデファクトスタンダードである Docker を対象とした SATD の目視調査を行う．調査の結果，Docker 内のコメントの内，約 3.4%が SATD に関する記述であった．また，バージョン固定に関する SATD や，PGP 等を用いた真正確認に関する SATD など，コンテナ仮想化技術固有の SATD を発見できた．
キーワード Self-Admitted Technical Debt, 技術的負債, コンテナ仮想化, Docker

1. はじめに

ソフトウェア開発において，開発者が意図的に混入させた場当たり的な解決策を，Self-Admitted Technical Debt（SATD）と呼ぶ [1]．SATD は将来への問題の先送りである「負債」の比喩であり，ソースコード中のコメントとして `TODO` という文字列と共に記載されることが多い．これまでに SATD に関する様々な研究が実施されており，その分類と体系化 [2] [3] や，SATD の自動検出 [4]，解消（返済）の実態調査 [5] [6] などが存在する．

他方，近年サーバ等のインフラを支える技術として，コンテナ仮想化と呼ばれる仮想化技術が注目を浴びている．コンテナ仮想化ではコンテナがホスト OS とカーネルを共有するため，ハイパーバイザ型の仮想化と比較して，CPU やメモリ等のシステム資源に対するオーバーヘッドを軽減できる．現在，コンテナ仮想化のデファクトスタンダードは Docker であり，様々なインフラに実利用されている．

この Docker においては，既存の SATD 研究で調査対象とされている一般的なプログラミング言語と同様，多数の SATD が存在している考えられる．Docker では，イメージ(注1)の構築方法は Dockerfile と呼ばれるファイルに手続き命令の集合として記述される．Dockerfile はインフラ構築の手順を示した一種のソースコードであり，開発者の場当たり的な実装や悩み，後回しタスクが SATD としてコメントの形で明記されている可能性がある．これら SATD の検出と分類により，場当たり的な実装による低品質な Dockerfile の検出や，Dockerfile 開発者が抱える実装の悩みの体系化，あるいはその負債解決方法のパターン化などが実現可能だと考える．

また，Dockerfile に混入された SATD は，他の Docker イメージへの波及につながる恐れがある．Docker では，あるイメージをベースとして新たなイメージを作る，イメージの継承という仕組みが存在する．よって，ベースとなったイメージの SATD は継承先のイメージに波及する．このような SATD による低品質実装の波及という問題を回避するためにも，SATD の調査，及びその検出方法が必要だと考えられる．

本研究の調査は，Docker における SATD の体系化を目的とする．その目的を達成するため，以下 3 つの Research Question（RQ）を設定し，Dockerfile のコメントの目視調査を行う．

RQ1: Docker にはどの程度 SATD が存在するか
RQ2: どのような種類の SATD が存在するか
RQ3: それぞれの SATD の割合はどの程度か

目視調査では，Dockerfile のコメントが SATD であった場合には系統毎に分類し，各種類の SATD の量や性質を調査する．

Docker の公式レジストリである，Docker Hub の人気上位 1,250 イメージを構築する Dockerfile のコメントの内 405 件を目視調査した結果，SATD は研究対象とした全コメントの約 3.4%であった．また，Docker の SATD は 5 種類の大分類と，それを更に詳細に分類した 11 種類の小分類に分類できた．そ

（注1）：Dockerfile によって構築されたコンテナの初期状態

の中には，バージョン固定に関する SATD や，PGP(注2)等を用いた真正確認に関する SATD など，Docker 特有の SATD の存在を確認できた．

2. 準　　備

2.1 技術的負債

1993 年 Cunningham によって，場当たり的な実装の比喩として技術的負債という概念が導入された [7]．その概念の導入以降，多くの研究者が技術的負債に関する研究を行っている．

Potder らは開発者が意図的に混入させた技術的負債についての研究を行い，それらの技術的負債を Self-Admitted Technical Debt （SATD）と名付けた [1]．彼らは，コード内のコメントを SATD の指標としてプロジェクト内にどの程度 SATD が存在しているか等の調査を行った．さらに，Bavota らは Potder らの研究 [1] の追実験を行い [3]，SATD を 6 個の大分類と，更に詳細な 10 個の小分類に分類した．一方，Liu らは研究対象としてディープラーニングのフレームワークに焦点を当てた SATD の研究 [8] を行っており，Java 以外のプロジェクトにおける SATD の蔓延を示した．また，Liu らはこの研究において 2 種類のドメイン固有な SATD を発見している．これらの研究を含め，これまでに多くの研究で SATD がソフトウェアに悪影響を及ぼすことが明らかにされている [9] [10]．

2.2 コンテナ仮想化技術

コンテナ仮想化は OS レベルの仮想化技術であり，コンテナと呼ばれる仮想的な環境を提供する技術である．現在は Docker がコンテナ仮想化のデファクトスタンダードとなっており [11]，その発表以降，コンテナ仮想化が急速に注目されるようになった．Docker の特筆すべき点の 1 つはコンテナの構築手順をスクリプトの形式で記述できる点にある．Docker では，Dockerfile というファイルからイメージを作成（ビルド）する機能がある．Dockerfile はスクリプト形式のテキストファイルであり，配布や共有が容易で，同じ Dockerfile をビルドすることで誰でも同じコンテナを作成できる．このことから，Docker はインフラツールとして，現在多くのアプリケーションに使用されている．

2.3 Docker における SATD

既存の SATD 研究 [3] [10] では，調査題材として Java が広く用いられており，ある SATD がオブジェクト指向の継承によって他クラスや他プロジェクトへ波及するという問題が指摘されている．Docker でも同様に，任意のイメージを継承する機能があるため，Docker ドメインにおいても SATD の波及という問題が発生する恐れがある．他方，ディープラーニングのフレームワークなど特定のドメインを対象とした SATD の研究は行われているが [8]，Docker を対象とした研究は行われていない．そのため，Docker における SATD の性質を明らかにするための調査や，波及の防止は重要な課題と言える．

本研究では 1. 章で示した 3 つの RQ に沿って調査を行う．下記に各 RQ の研究動機を示す．まず, 本研究は Docker の SATD の実態を明らかにする基礎となる研究であるため，Docker に

(注2)：Pretty Good Privacy の略．暗号ソフトウェアの一種である．

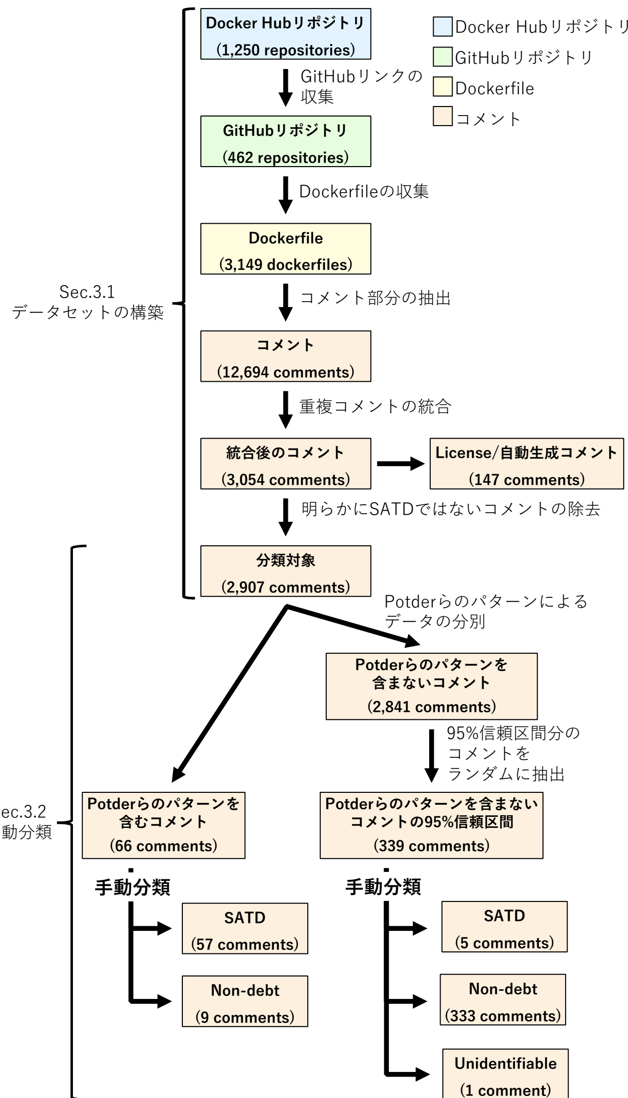

図 1　調査手法の概要

は SATD が存在するかを調査する（RQ1）．Docker に SATD が存在すれば，それに対する解決策が必要となる．その解決策は全ての SATD に共通していないため，異なる種類の SATD には異なる解決策が必要である．そこで，Docker にはどのような種類の SATD が存在するかを調査する（RQ2）．さらに，RQ2 で最も多く確認された SATD は，多くのイメージ開発者が苦悩していることである可能性が高い．そのため，それぞれの Docker の SATD の割合はどの程度かを調査する（RQ3）．

3. 調 査 手 法

本章では，データセットの構築から SATD の手動分類までの本研究の調査手法を説明する．図 1 に調査手法の流れを示す．図 1 では Docker Hub を起点としたデータセットの構築方法（3.1 節）から，SATD の手動分類（3.2 節）までの手法の流れを示している．図中の四角は各手順におけるデータの集合を表している．色によってその単位が異なる点に注意されたい．また，括弧内は収集できたリポジトリやコメントの数を表している．以降では本図の流れに従って調査手法を説明する．

3.1 データセットの構築

3.1.1 対象プロジェクトの選定

本研究では Dockerfile 内に存在するコメントを調査の対象とする．そのため，Dockerfile のコメントを収集したデータセットを作成する必要がある．SATD は開発の過程で混入すると考えられるため，継続的に開発やメンテナンスが行われているプロジェクトの Dockerfile を調査対象とするべきである．そこで本研究では，Docker イメージがプロジェクトの主体であり，Docker の公式レジストリである，Docker Hub の人気リポジトリの Dockerfile からデータセットを構築することを考えた．さらに，Docker Hub の人気イメージは他のイメージへの継承の機会も多く，Docker Hub の人気イメージに SATD が残っていると，多くのイメージにその影響が波及するため，SATD の返済の重要度が高いと考えられる．

しかし，Docker Hub には直接 Dockerfile が設置されておらず，GitHub など外部サービスを利用して版管理などが行われており，そのリポジトリへの URL が Docker Hub リポジトリ内に記載されていることが多い．本研究では Docker Hub の人気上位 1,250 リポジトリを対象として，そのプロジェクトに紐づけられた GitHub リポジトリへのリンクを収集した結果，462 個のリンクを収集できた．さらに，そのリンク先のリポジトリ内の，Dockerfile を収集した結果，3,149 個の Dockerfile が得られた．それらの Dockerfile のコメント部分のみを抽出した結果，12,694 件のコメントが得られた．

3.1.2 コメントの統合

Dockerfile を扱う多くの GitHub リポジトリでは同一リポジトリ内に，OS やソフトウェアのバージョンや Linux ディストリビューションのみが異なるイメージを作成する Dockerfile が多く存在する．例えば PostgreSQL では，異なる 6 つのバージョンの PostgreSQL をインストールしたイメージがそれぞれ別の Dockerfile で管理されている．さらにそれぞれのバージョンにおいて，2 種類のディストリビューションに応じた Dockerfile も個別に管理している．よって上記の場合，計 12 種類の Dockerfile がリポジトリ内に設置されている．(注3) これらの Dockerfile はほぼ内容が一致しており，SATD 調査に対するバイアスを生む要因となる可能性がある．実際にデータセット内のコメントには同一リポジトリから全く同じ内容のコメントが多数抽出されていた．そこで，それらのコメント同士を同一コメントとして扱うため，単一のコメントに統合した．その結果，データセットのコメント数を 3,054 件に集約できた．

3.1.3 不要コメントの除去

Dockerfile のコメントにはライセンスや自動生成ツールによって挿入された明らかに SATD ではないコメントが含まれる．そのため，データセットの中から，ライセンスと自動生成コメントの内，Potder らが発見した SATD に含まれやすい 63 のパターン（`fixme`，`hack` など，以降パターンと呼ぶ(注4)）[1] を含まないコメントを除去した．その結果，147 件のコメントを除去し，分類対象の 2,907 件のコメントを得られた．

(注3)：Docker の文化として広く用いられている管理方法であり，これによりイメージの利用者が自由にバージョンやディストリビューションを選択できる．

(注4)：厳密には 63 パターンに `TODO` という文字列を追加した．`TODO` は SATD の重要なキーワードだが，63 パターンには含まれていなかったためである．

3.2 手 動 分 類

Phase 1: SATD に対する合意の形成

Docker の SATD を分類する前に，既存の研究ではどのようなコメントをどの種類に分類しているかを理解し，それぞれの SATD の分類基準の合意を分類者の間で形成する必要がある．そのため，Maldonado らが Java のコメントを分類して作成したデータセット [4] を使用する．そのデータセットは 10 個の Java プロジェクトから収集されたコメントで形成されており，それぞれのコメントが 4 種類の SATD と Non-debt に分類されている．本研究の手動分類は第 1，第 2，第 3 著者の 3 名の著者によって独立に行われた．Phase 1 では，そのデータセットから各種のコメントを 20 件ずつランダムに抽出した．続いて，その分類について議論し，それぞれの SATD の明確な判断基準を作成した．

Phase 2: Docker への SATD 分類の適用と分類基準の定義

手動分類にあたって，まず分類対象となる 2,907 件のコメントを SATD を含む可能性が高いか否かで自動的に分類する．この目的は，SATD を含む可能性の高いコメントを優先的な目視確認と，Phase 1 での Java を題材とした負債認識の合意形成の Docker ドメインへの拡張である．この自動分類は，コメントにパターンが含まれているか否かによって行った．その結果，パターンを含むコメントが 66 件，パターンを含まないコメントが 2,841 件であった．Phase 2 ではパターンを含むコメントとパターンを含まないコメントをそれぞれ 50 件ずつ，合計 100 件のコメントを著者 3 名が独立に分類し，それぞれの著者が行った分類結果を元に議論を行った．議論の結果，オブジェクト指向言語を題材として作成された Maldonado らの SATD 分類基準 [4] は，Dockerfile のコメントに対して完全には適合しないことが明らかとなった．そのため，Bavota らの分類 [3] を参考にして Dockerfile のコメントに適合する分類基準を定義した．

Phase 3: 目視確認による SATD の分類

Phase 2 での分類の結果，パターンを含むコメントでは約 8 割が SATD であった．そこで，それらは SATD である可能性が高いと考えられたため，66 件全てを手動分類の対象とした．対して，Phase 2 で分類したパターンを含まないコメントには SATD が存在しなかったため，それらは SATD である可能性が低いと考えられた．また，その総数は 2,841 件であり，これらの全ての分類は困難であるため，そこから誤差 5%の 95% 信頼区間を満たす量をランダムに選択し，339 件を手動分類の対象とした．そこで Phase 3 では，Phase 2 で未確認のコメントを対象としてパターンを含むコメントと含まないコメントのそれぞれ 16 件と 100 件を手動分類の対象とした．以上の分類結果から，議論を通し，分類基準の調整を行った．

Phase 4: 手動分類の完遂と再分類

最後に 339 件のパターンを含まないコメントの内，未確認である 189 件を著者 3 名が独立に分類した．Phase 4 でも Phase 3 と同様に，議論を通して分類基準の再調整を行った．ここで，

Phase 3, 4 の議論で分類基準の調整を行ったため，これまでの分類で分類の揺れが生じている可能性があった．そのため，分類に揺れが生じている可能性があるコメントを対象に議論を行い，一部のコメントを再分類した．

4. 調 査 結 果

4.1 RQ1: Docker にはどの程度 SATD が存在するか

手動分類の結果，その対象のコメントの内，パターンを含むコメントでは 57 件（86.4%）が SATD であった．一方，パターンを含まないコメントでは 5 件（1.5%） が SATD であった．パターンを含まないコメント全体の 2,841 件の中にも同等の割合で SATD が含まれていると仮定すると，パターンを含まないコメント全体では約 42 件が SATD であると考えられる．したがって，分類対象の 2,907 件のコメント全体では合計で約 99 件の SATD が存在し，その割合は約 3.4% である．この結果から，Docker ドメインの SATD の存在が明らかになった．

また，パターンを含むコメントの結果に注目すると，Docker でのパターンが SATD を判別する精度は 86.4%である．一方，パターンに一致しないコメント全体の SATD 件数が 42 件であったと見積もると，分類対象 2,907 件のコメントに対する SATD 検出の再現率は 57.6%となる．これらの数値から F 値を算出すると 0.69 となる．したがって，Docker ではパターンの使用により高い精度で SATD を検出可能である．

4.2 RQ2: どのような種類の SATD が存在するか

手動分類において，SATD に分類した全 62 件のコメントを 5 個の大分類と，11 個の小分類に分類した．この分類軸は Bavota らの分類 [3] に基づいて手動分類を行った著者 3 名の合意をもって作成した．その分類木と分類結果を図 2 に示す．図中の SATD というルートから分岐した 5 個のノードがそれぞれ大分類を表し，大分類からさらに分岐したリーフはその大分類を詳細に分類した小分類を表す．

また，表 1 に本研究で適用した小分類の定義を示し，その詳細を以下に示す．

Code debt: Code debt はコードの保守を困難にする恐れのある SATD である．本研究では，Code debt を Workaround（14 件），Missing functionality（9 件），Base image（5 件），Version（5 件）の 4 つの小分類に分類した．

Code/Workaround は次善策による実装に関する SATD であり，全小分類の中で最も多く確認できた．この SATD では，より最適な手法の存在を自覚していながら時間的要因などで次善的な実装をしているため，後に改善すべきであるという旨のコメントが多かった．

Code/Missing functionality は外的には観測できないコンテナ内部の処理の欠如に関する SATD である．この SATD は，未実装が致命的になるような機能ではなく，付属的で実装の優先順位が低く設定されるような機能に対しての言及が多い．

Code/Base image は自身のイメージで継承するベースイメージのバグに対する注意喚起を行っているか，ベースイメージ自体の変更を求める SATD である．この SATD はベースイメージという Docker 特有の概念に関わる SATD であるため，Docker 特有の SATD であると考えられる．

Code/Version はパッケージマネージャや Git で取得するソフトウェアやツールのバージョン固定を促す SATD である．Docker では公式のベストプラクティスとして，コンテナ内でダウンロードするソフトウェアのバージョン固定が推奨されているため [12]，この種の SATD が存在すると考えられる．

Test debt: Test debt はテストやコンテナの検証に関する SATD である．本研究では，Test debt を Integrity（8 件），Improvement for test（3 件）の 2 つの小分類に分類した．

Test/Integrity はコンテナ内で利用するバイナリファイルやハッシュ値の真正確認の不足に関する SATD である．通常このような真正確認は PGP や Linux の sha256sum コマンドなどによって行われる．Docker では外部のファイルやソフトウェアの利用が必要になる場合が多いため，この SATD が多く確認できたと考えられる．そのため，Code/Base image 同様，Test/Integrity も Docker 特有の SATD であると考えられる．

Test/Improvement for test はテスト手法の改善を求める SATD である．その改善はバグ修正のためではなく，テストの効率化やテスト部分の保守性向上のための改善として定義した．

Defect debt: Defect debt は時間的制限や解決の優先度が低いなどの事由で完全な解決を先送りにされているバグに関する SATD である．本研究では，Defect debt を Hack（4 件）と Latent（2 件）の 2 つの小分類に分類した．

Hack は外部システムに存在しているバグを回避するための措置に関する SATD である．Docker では外部システムの使用が多く，そのシステムにバグが含まれていた場合には，そのバグを回避するための対策が必要である．しかし，そのバグへの対策により，一時的な環境改変が施されている場合があるため，この SATD が存在するイメージは慎重に扱う必要がある．

Latent はそのイメージ内に含まれる潜在的，あるいは将来的に起こりうるバグに関する SATD である．この SATD は現時点でイメージに悪影響を及ぼさないが，外部システム側の後方互換性を持たないアップデートなどにより，現状使用しているコマンド等が使用不可能になり，発生するバグである．

Design debt: Design debt はデザインパターンに反した設計に関する SATD である．Docker は Java のようなオブジェクト指向言語ではないため，そのデザインパターンも異なっている．そのため，Bavota らが Java で行った分類 [3] と完全に合致する分類は存在しなかった．本研究の調査で発見した Design debt は全て Size reduction（4 件）に分類できた．

Design/Size reduction はイメージサイズの削減のための工夫を求める SATD である．Docker ではイメージの pull や push のコストを削減するため，イメージのサイズを可能な限り小さく保つことが望ましいとされている [12]．そのため，このような SATD が混入したと考えられる．この SATD は実行に直接影響を与えないため，対応が後回しにされやすいと考えられる．

Process debt: Process debt はデプロイや Dockerfile のレビューなど，開発の特定のプロセスにおける問題に関する SATD

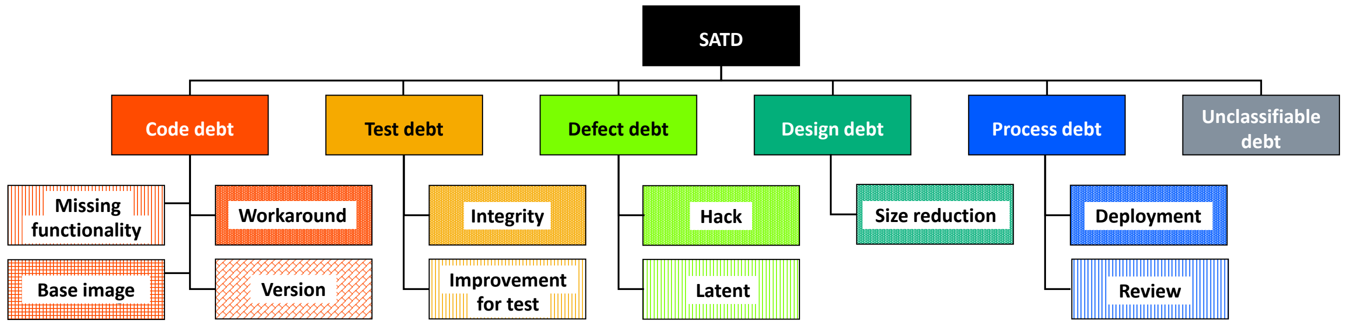

図 2　本研究で用いた SATD の分類木

である．この SATD はデプロイなど，Docker のインフラツールとしての側面に深く関わりがあるため，Bavota らの Java の SATD 分類には存在していない，Docker 特有の SATD である．本研究では，Process debt を Deployment（2 件）と Review（2 件）の 2 つの小分類に分類した．

Process/Deployment はデプロイの際に起こりうる問題に関する SATD である．2.2 節の通り，Docker は様々なソフトウェアのインフラツールとして使用されている．加えて，アプリケーションを Docker コンテナにデプロイする開発者も多く，開発環境と本番用の環境に同一のコンテナを使用する開発者も存在する．そのため，デプロイに関して特別な注意が必要となる場合が生まれると考えられる．

Process/Review は Dockerfile 自体のレビューを求める SATD である．この SATD は外部システム等の検証ではなく，Dockerfile やイメージ自体についてのレビューを他の開発者に求める SATD である．Docker は 2013 年に公開された比較的歴史の浅い技術であるため，開発者自身にノウハウが蓄積されておらず，このような SATD が生じたと考えられる．

Unclassifiable debt: Unclassifiable debt は何かに対する要求が示されているため負債であると考えられるが，その記述の曖昧さ等により，分類ができなかった SATD である．本研究で確認された Unclassifiable debt は全て `TODO` という文字列を含んでおり，確実に何かに対する要求が示されていた．分類の上で，著者らはその SATD が混入した時点まで版管理の履歴を遡るなどして，コメントの意味の究明に努めたが，何に対する要求が示されているかを解明できなかった．

以上より，本調査で発見された SATD は Unclassifiable debt を除いて，5 個の大分類に分類でき，それをさらに 11 個の小分類に分類できた．加えて，それらの分類の中には他の言語では発生しない，Docker 特有の SATD の存在を確認できた．

4.3　RQ3: それぞれの SATD の割合はどの程度か

図 3 に全 SATD に占める大分類と小分類それぞれの SATD の割合を表す．2 本の帯グラフの内，上側が大分類，下側が小分類のグラフを表している．

調査の結果，大分類では Code debt が最も多く，全 SATD の 53.2%であった．次いで 17.7%の Test debt が続いた．Maldonado らの Java を対象とした SATD 分類の結果では Test debt が全 SATD の 2%程度であった [4] ことに対し，Docker では 20%近くを Test debt が占めていたため，Docker ではテストに関する実装が後回しにされやすいと考えられる．一方，Docker では発生しない SATD の存在も確認できた．

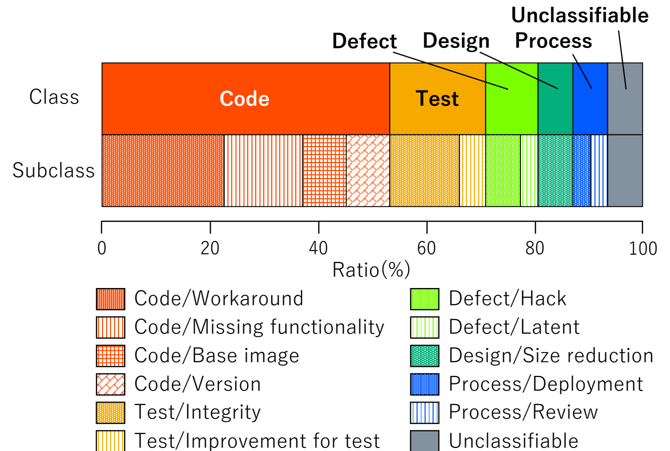

図 3　それぞれの SATD の割合

また，小分類に注目すると，全小分類中最も多かったのは Code/Workaround で，全体の 22.5%を占めていた．次いで Code/Missing functionality や Test/Integrity が多く確認された．したがって，イメージ開発者の多くが Docker における様々な機能の最適な実装手法や真正確認の手法などに悩みを抱えて

表 1　分類の定義

分類名	定義
Code/Workaround	次善策での実装に関する負債
Code/Missing functionality	内部処理の欠如に関する負債
Code/Base image	利用ベースイメージに関する負債
Code/Version	特定バージョンへの固定に関する負債
Test/Integrity	利用バイナリの真正確認に関する負債
Test/Improvement for test	テストの改善に関する負債
Defect/hack	外部システムのバグに関する負債
Defect/latent	潜在バグに関するの負債
Design/Size reduction	イメージのサイズ削減に関する負債
Process/Deployment	デプロイに関する負債
Process/Review	Dockerfile のレビューに関する負債
Unclassifiable	負債ではあるが分類不可な負債

いると考えられる．加えて，Docker 特有の SATD の割合を合計すると，37.1%となり，本調査で確認できた SATD の約 1/3 が Docker 特有の SATD であったと言える．

5. 考　　察

5.1 通常のプログラミング言語の SATD との比較

4.3 節で述べた通り，Docker の SATD は Java の SATD と比較して，その Test debt の割合が高いと考えられる．その理由として，Java 等のプログラミング言語では開発者自身が書いたコード部分をテストするのに対し，Docker では外部から取得してきたバイナリファイルの真正確認や，外部システムがそのイメージで動作するかのテストなど，多様なテストや検証を要している．また真正確認においても，その手法は一元化されておらず，PGP や GnuPG など様々なソフトウェアを使用して行う必要がある．そのため，テストの実行に Java にはない難しさが存在していると考えられる．

続いて，Defect debt に着目する．Java 等の他の言語で作成したアプリケーションにおける Defect debt はそのアプリケーションを構築するために，開発者自身が書いたコード内に生じたバグへの言及が多い．一方 Docker では，Docker イメージを構築する上で，外部のシステムやツールを多く利用する必要がある．そのため，イメージの開発者には打つ手がない外的要因に左右されやすく，Docker には Defect/Latent のような SATD が存在すると考えられた．

5.2 知　　見

まず，この研究が Docker の研究者に提供する知見は，Docker における SATD の存在を明らかにしたことである．これにより，その返済や検出に関する研究の重要性が高まったと考えられる．例えば，数年後に本研究で発見された SATD を追跡し，Docker の SATD がどの程度の期間で返済されるか調査する研究を行える．さらに，相当数の Docker の SATD を収集できれば，機械学習などを用いた自動検出に関する研究が行える．

続いて，Docker の開発者に提供する知見は，多くの Docker イメージの開発者が悩みを抱えていると考えられる点の解明である．4.3 節で述べた通り，多くの開発者は最適な実装手法や真正確認などの手法について悩みを抱えている傾向にあると推察される．そのため，ベストプラクティスなど公式な文書に，多くの開発者の目に止まる形で，最適な実装手法のモデルケースや真正確認の必要性を示すべきであると考える．

最後に，Docker イメージの開発者に提供する知見は，Docker の SATD にはどのような種類が存在し，どの種類の SATD が多く存在するかを明らかにしたことにある．これにより，イメージの開発者は Docker イメージを構築する上で留意すべき点が明確になり，新たな SATD の混入を予防し，その SATD が多くのイメージに波及することを防げる．

6. お わ り に

本研究では，Docker における SATD の実態調査として，Dockerfile のコメントに存在する SATD の分析を行った．その結果，Docker ドメインでも Java と同様に SATD が存在し，Dockerfile のコメントには SATD が約 3.4%存在することを明らかにした．それらの Docker の SATD は 5 種類の大分類の 11 種類の小分類と分類でき，それらの中には Docker 特有の SATD が存在していた．

今後の展望としては，キーワードでは検知できない Docker の SATD を機械学習などの自動的な手法で検出する研究が挙げられる．一方，本研究は Docker の SATD を理解するための初歩となる役割を果たしているが，Docker の SATD の返済については明らかになっていない．また，SATD の返済に関する研究は多くのイメージの開発者にその返済パターンを提供できることが期待できるため，重要な課題であると考える．

謝辞　本研究の一部は，JSPS 科研費 JP18H03222，および，JSPS・国際共同研究事業の助成を得て行われた．

文　　献

[1] A. Potdar and E. Shihab, “An Exploratory Study on Self-Admitted Technical Debt,” In Proc. IEEE International Conference on Software Maintenance and Evolution, pp.91–100, 2014.

[2] E. d.S. Maldonado and E. Shihab, “Detecting and quantifying different types of self-admitted technical Debt,” In Proc. International Workshop on Managing Technical Debt, pp.9–15, 2015.

[3] G. Bavota and B. Russo, “A Large-Scale Empirical Study on Self-Admitted Technical Debt,” In Proc. IEEE/ACM Working Conference on Mining Software Repositories, pp.315–326, 2016.

[4] E. d.S. Maldonado, E. Shihab, and N. Tsantalis, “Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt,” IEEE Transactions on Software Engineering, vol.43, no.11, pp.1044–1062, 2017.

[5] E.D.S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, “An Empirical Study on the Removal of Self-Admitted Technical Debt,” In Proc. IEEE International Conference on Software Maintenance and Evolution, pp.238–248, 2017.

[6] S. Mensah, J. Keung, J. Svajlenko, K.E. Bennin, and Q. Mi, “On the Value of a Prioritization Scheme for Resolving Self-Admitted Technical Debt,” Journal of Systems and Software, vol.135, no.C, pp.37–54, 2018.

[7] W. Cunningham, “The WyCash Portfolio Management System,” SIGPLAN OOPS Mess., vol.4, no.2, pp.29–30, 1992.

[8] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks,” In Proc. International Conference on Software Engineering, 2020.

[9] N. Zazworka, M.A. Shaw, F. Shull, and C. Seaman, “Investigating the Impact of Design Debt on Software Quality,” In Proc. Workshop on Managing Technical Debt, pp.17–23, 2011.

[10] S. Wehaibi, E. Shihab, and L. Guerrouj, “Examining the Impact of Self-Admitted Technical Debt on Software Quality,” In Proc. International Conference on Software Analysis, Evolution, and Reengineering, pp.179–188, 2016.

[11] Open Container Initiative, “Why have all of these companies come together?,” accessed 2020-09-08. `https://opencontainers.org/faq/#why-have-all-of-these-companies-come-together`.

[12] Docker Documentation, “Best practices for writing Dockerfiles,” accessed 2020-09-17. `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`.