技術的負債:
ソフトウェア進化阻害要因の現在のとらえ方

小林 隆志　林 晋平　斎藤 忍

技術的負債は，ソフトウェア開発において開発コストを優先するために，最適でない解法を採用することを指しており，ソフトウェア進化を阻害する要因のひとつとして知られている．技術的負債の新しい点に，全ての進化阻害要因を解消すべき悪影響とみなすわけではなく，一定の負債を受け入れ悪影響とならないよう管理するものとして取り扱う点がある．本論文では，こういった達成阻害要因のとらえ方の発展も含め，これまでに議論されてきた技術的負債およびその周辺技術について概観し，技術的負債は設計，コードの品質に関するこれまでの技術とどのような関係にあるのかを解説する．

Technical debt refers to the adoption of suboptimal solutions in software development in order to prioritize development costs, and is known as one of the barrier factors that hinder software evolution. A new aspect in technical debt is that not all of the debts are regarded as negative effects to be resolved, but rather certain debts are accepted and managed to prevent them from becoming negative effects. This paper provides an overview of technical debt and related technologies, and explains how technical debt relates to previous technologies related to design and code quality.

1　はじめに

技術的負債という表現を，研究論文だけでなく技術的な記事などでも見かけることが多くなった．これらは単に長く保守が続けられている大規模システムにおいて使用されている古い技術などを指している場合もあるが，ソフトウェア工学分野で議論される技術的負債はより広い意味を含む概念である．

技術的負債は，ソフトウェア開発において開発コストを優先するために，最適でない解法を採用することを指す (詳細は 2 節)．この解法の選択は，将来の開発において開発コストを増大させ，ソフトウェア進化を妨げる原因となり得る．顧客のニーズや利用環境の変化に迅速に対応する必要がある現在のソフトウェア開発においては，これまでに比べ短い期間で継続的にリリースを行うことが求められており，技術的負債を発生させることで開発期間を短縮することと，技術的負債による影響でソフトウェア進化が困難となることのトレードオフを検討する必要がある．このことは，ソフトウェア開発における設計判断の原則 [65] などでも議論されている．

ソフトウェア工学の研究コミュニティでは，技術的負債の研究が 2010 年ごろから徐々に活発になっている．2010 年から開始され，ソフトウェア工学分野の国際会議 ICSE などに併設して開催されてきたワークショップ International Workshop on Managing Technical Debt (MTD) が 2018 年から国際会議 International Conference on Technical Debt (TechDebt) に拡大されるなど，多くの研究者の関心を集めている．

技術的負債に関しては，個々の手法提案や分析報告だけでなく，それらに対する系統的マッピングによる論点の整理 (Systematic Mapping Study) [4] [19] [37] [48] やサーベイ論文 [1] [5] [7] [8] [30] [34] [66] が多く発表され

Technical Debt: The Current Understanding for the Barrier to Evolve Software

Takashi Kobayashi, 東京工業大学, Tokyo Institute of Technology.

Shinpei Hayashi, 東京工業大学, Tokyo Institute of Technology.

Shinobu Saito, 日本電信電話株式会社, Nippon Telegraph and Telephone Corporation.

ている．さらに，これらのサーベイ論文の分析による，技術的負債に関する共通認識の形成や不足している研究領域の検討もなされている[49]．

ソフトウェア進化を阻害する要因については，技術的負債が注目される以前から多くの研究が行われており，既存概念や支援技術が存在する．技術的負債は既存の進化阻害要因の考え方と異なり，全てが悪影響とみなされるわけではなく，一定の負債を受け入れ悪影響とならないように管理するものとして取り扱われる点が新しい．しかしながら，前述のサーベイ論文等は技術的負債に関する論文を技術的負債の観点で整理分類するものであり，既存の研究や支援手法との関係が議論されていない．具体的には，不吉な臭いやアンチパターン (詳細は 4 節) といった既存の類似概念及びそれらを扱う研究との対比の観点が欠けている．この点を補うことにより，進化阻害要因のとらえ方の発展を明らかにするとともに，技術的負債というとらえ方と既存の概念との違いが明確になると考える．

本論文では，これまでに議論されてきた技術的負債およびその周辺技術について，2018 年に発表された文献[49] 以降の研究成果を取り入れながら概観する．さらに，技術的負債がソフトウェア進化の阻害要因やそれに対する支援手法とどのような関係にあるかを解説する．

以下では，まず 2 節で技術的負債の定義を紹介し，技術的負債の発生経緯に基づいた分類について説明する．次に 3 節で，技術的負債がソフトウェア進化の阻害要因とならないように行うべき技術的負債の管理について解説したあと，4 節で既存の進化阻害要因に関係する概念や技術との関係を説明する．また，5 節で，成果物や開発工程の観点で種類分けされた技術的負債について解説し，特に研究事例の多い設計に関する技術的負債とコードに関する技術的負債の周辺技術について説明する．最後に，6 節で本論文をまとめる．

2 技術的負債

2.1 定義

ソフトウェア開発に関係する技術的負債とは，開発の過程で対応を後回しにした懸念・問題点を指す

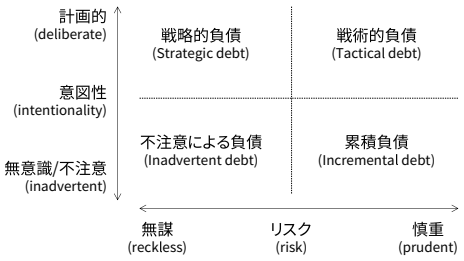


図 1　リスクと意図性による技術的負債の分類．[18][22] の分類図に Tom ら[66] の分類定義を加えたもの．

[13]．最適であるが開発コストが必要な手法をとる代わりに，最適でないが開発コストの低い簡単な手法をとったことで，のちに生じる追加作業のコストの比喩的表現[31] として負債という用語を用いている．Cunningham が 1992 年に比喩として言及[13] したものが最初の登場とされている[18]．また，McConnell は技術的負債を，短期的な視点では有効な応急措置だが，長期的に考えると複雑度が増し開発コストが増加する設計や実装手段と定義している[40]．

負債の名をとっているように，経済活動における負債と類似する点が多い．本来かかる開発コストを省くことで開発の時間を短縮している，つまり将来から時間を借りている (借入)．一方で，長期的に対象システムを開発する過程では，先送りにした作業を後に行わなければならない (返済)．この作業が後回しになればなるほど，その作業を行う追加コスト (利息) が大きくなる．これらが，変化する顧客要求に迅速に適応する必要がある現在のソフトウェア開発において発生しやすい状況であることから，2010 年ごろから用語として普及してきた．

2.2 技術的負債の分類：リスクと意図性

技術的負債はリスク (risk) と意図性 (intentionality) の二つの観点で分類できる (図 1)[18][22]．前述のように，負債を返済する際には利息となる追加コストが発生する．リスクは，将来的にその負債が予想される利息分を含めて返済可能であるか，つまり技術的負債を発生させたことの影響を慎重 (prudent) に検討しているか，しておらず無謀 (reckless) ともいえる状態であるかの観点である．意図性は，計画的

(deliberate) に負債を発生させたのか，無意識あるいは不注意 (inadvertent) で発生させたかの観点である．

Tom らはこれらの観点の組合せについて 4 種類に分類し，以下のように命名している [66]．

Strategic debt (戦略的負債) [無謀，計画的] 計画的に導入した大型の長期借入の技術的負債を指すものである．例えば，他社に先駆けて市場に出ることを目的とした場合のように，短期的な開発速度を上げるために，長期的なソフトウェア問題の可能性を無視するような状況を指す．

Tactical debt (戦術的負債) [慎重，計画的] 戦略的負債と同様に，計画的な技術的負債である．Strategic debt との違いは，短期的な借り入れを意図している点で，例えば，最適でない解法の選択や，アップグレードが必要なライブラリやフレームワークの使用など，返済するための作業がタスクとして明確に特定できるものを指す．

Incremental debt (累積負債) [慎重，無意識] リスクは小さいが無意識に発生させる，もしくはそのリスクの小ささから見逃されてしまう負債を指す．例えば，一般的すぎる変数名を付ける，不十分なコメントを書く，分割して作るべき機能を凝集度の低い 1 つのクラスとして実装する，コーディング規約を守らない，などの小さな作業の先送りを指す．このような負債は，それぞれは軽微なリファクタリング活動で返済でき，単体では大きな作業コストの増加とならないが，累積により影響が増大し，簡単なリファクタリングでは解決できない状況に発展する．Tom らは，累積負債をクレジットカードの負債に例えて紹介している [66]．

Inadvertent debt (不注意による負債) [無謀，無意識] 意図せずに発生させた技術的負債のことで，理解不足や見落としによるものを指す．開発者が負債を発生させた自覚を持たないため，リスクの高い無謀な負債となる．

これらの分類のうち，最も避けるべきものは不注意による負債である．戦略的負債は，意図的に発生させた負債であり，返済時の負担の想定が不十分であったとしても，不注意による負債よりは安全である．戦術的負債は，負債を追うリスクとのトレードオフを考慮したうえで意図的に発生させるものであり，4 種類の中で最も許容できる．累積負債は，不注意による負債と同様に開発者が意図せず混入させる負債であるが，早い段階であれば小さな追加コストで返済が可能である．

なお，上記の分類の説明からも分かる通り，技術的負債を発生させることは必ずしも悪影響を生むだけではない．技術的負債は前述のとおり，本来かかる開発コストを省くことで開発の時間を短縮する効果を持つ．過度に技術的負債を避け開発を鈍化させるよりは，負債として開発コストを借り入れることが最適となる状況も少なくない [18]．また，経済活動における負債との違いとして，他のシステムへ移行するなどして，負債を返済することなくそのソフトウェアが役目を終える場合がある †1．

また，この 2 観点に基づく分類が難しい技術的負債として，負債を発生させたタイミングでの最適解と現在の状況が乖離してしまい，結果として大きな負債となってしまうケースがある．これは，戦術的負債として発生させた場合だけでなく，技術的負債を生み出さないために最適な解を選択したものの当初の想定と異なる方向に開発が進んだ場合にも起きうる．例えば，反復開発により段階的に機能を提供していく場合や，長期間の保守を経て関連システムとの接続性や相互運用性が変化する場合が該当する．特にアジャイルソフトウェア開発を適用している場合に発生する技術的負債やその影響と技術的負債の管理の方法については，Behutiye らがサーベイ論文 [7] にまとめている． また，Change Anything Changes Everything (CACE) の原理として知られる機械学習システム固有の特徴から，技術的負債が急速に肥大化する問題 [54] も同様にリスクと意図性では分類できない負債であると考える．

Tan らは発生させた開発者が後に自ら返済した負債に着目し，負債発生の原因と返済行動の動機について分析を行っている [63]．実際の開発者から回収した

†1 そのソフトウェア自身もしくは一部が他の用途に転用・再利用できるという資産としての議論はここでは含まない．

181 件のアンケート回答を分析した結果，開発期間や開発チームの負荷状況の制約によって意図的に発生させた戦術的負債の言及が多く，開発者はその負債に対して利息が大きくなる前に責任をもって返済行動をとることが多かったことが報告されている．このように，意図的に発生させた技術的負債に関して最も重要な点は，その検討内容や判断の結果を記録として残すことである．これらの記録が欠損してしまうと返済にかかる開発コストの見積もりやリスクそのものが把握できず，不注意による負債と同様にリスクの大きい無謀な負債となる．

負債を発生させるかどうかを決断する際のリスク検討が困難もしくは検討が結果として不十分であった場合にも，リスク検討をしなかった場合と同様に，現在の負債の影響を把握することが重要となる．このため，技術的負債の発生を記録し負債の状況を監視する技術的負債管理が重要となる．技術的負債の管理については 3 節でより詳しく説明する．

2.3 技術的負債の分類: Self-Admitted Technical Debt

技術的負債を分類するうえでの別の観点として，いつ，誰が，その負債の存在を認めたかがある．技術的負債は，将来の返済を想定して低コストの代替手段を選択することで生じるものであるため，開発者がその存在を認め，記録を残して管理を試みたりする．例えば，意図的な混入の際に，開発者はコードコメントや課題管理システム上の課題 (Issue)，コミットログ [36] などに `TODO`，`FIXME`，`HACK` などといったラベルを付けた軽微なメモとして，一時的な問題回避を行ったことや，対応が不足したままであることを残す場合がある．これらは，開発者がより良い解に気が付いていながら開発コスト等の観点で選択しないと判断する場合と，現在の解が最適ではない認識があるものの最適な解を思いつかない場合の双方で起こる．

このようなメモの付与を，負債を発生させた開発者による負債の自認および管理の行為とみなし，このように明示的に認識・管理された技術的負債を Self-Admitted Technical Debt (SATD) と呼ぶ[†2]．コードコメント等からのテキスト分析により比較的容易に検出・分析できることも助け，SATD は他の技術的負債と区別して活発に研究がなされている [57]．

初期の研究では，`TODO` など特定の語を含むソースコード中のコメントを対象に SATD の検出や分析が行われてきた．Potdar らは 4 つの大規模オープンソースソフトウェア開発のリポジトリに対して 62 個の検出パターンを定義して SATD を検出し，SATD の量および発生した理由を分析している [46]．Sierra らによる SATD のサーベイ [57] では，パターンによる検出手法以外にも，自然言語処理に基づく手法やアンサンブルテキストマイニングを適用する検出手法がこれまでに提案されており，アンサンブルテキストマイニングを用いる手法が最も精度が高かったことが述べられている．

また，Fucci らの報告 [24] では，`TODO` などの SATD コメントがどのように挿入されたかを調査している．調査したプロジェクトにおいて，84%から 100%の SATD コメントはソースコードの変更とともに挿入されており大部分は確かに開発者の自認行為 (Self-Admit) であったこと，残りのケースでは挿入を行った開発者が挿入対象のファイルの所有者 (owner) であった割合は特に低くなくプロジェクトごとの中央値は 10%から 42%の間であったことが報告されている．これは，SATD コメントは，負債を発生させた開発者による自認としてだけでなく，コードレビューの結果として他の開発者によって記録されたものも含む可能性を示している．

SATD の調査・分析の多くは，ソフトウェア開発成果物のうち主にソースコードを対象としている．一方で，5 章で後述するように，Requirements Debt や Design Debt など，実装工程以外でのソフトウェア開発成果物の上での技術的負債を考えることもできる．しかし，こういった成果物に対する調査・分析は行いづらく，また著者らの知る限り存在しない．その理由として，ソースコードに比べ，そもそも要求仕様書や

†2 ただし，研究によっては技術的負債がこういったメモの形で明示的に自認された事象そのものを SATD と呼んでいることもある．

設計仕様書の実例の収集が難しいことがあると考える．またそれに加えて，SATD はコメントという形でソースコードに埋め込まれる．これは，プログラムの振る舞いを記述するソースコードに対するメタ情報とみなすことができる．主に自然言語により記述される要求仕様書や設計仕様書においては，メタ情報を埋め込みやすいフォーマットが限定されてしまうため，SATD が含まれうる文書を収集することはさらに難しくなる．もっとも，要求分析工程や設計工程に関連する負債が実装工程に影響し，ソースコードに対する SATD として前工程での負債を記録することはあり，これらの検出や推薦の試みはある [70] [72]．

3　技術的負債の管理

3.1　技術的負債の管理に必要な活動

前述のように，技術的負債を適切に取り扱うためには，無意識な技術的負債の発生を防止すること，既に発生している技術的負債を追跡し負債が返済可能なレベルであることを把握する必要がある [2]．すなわち，新たな技術的負債が発生しても，過度に悪影響でないなら無理に対応せず把握するに留めておく管理活動も重要となる．これまでもソフトウェア進化を阻害する要因として，成果物や開発組織の品質の観点での議論が多数行われてきたが，こういった管理の考え方は技術的負債で導入されたものである．

文献 [37] では，このように技術的負債を管理する活動を 8 種類に分類しており，この分類は多くの論文で引用されている．文献 [49] では，8 種類のうち「計測」と「順位付け」を監視活動を構成する活動としてグループ化し，技術的負債の管理を図 2 に示すように「防止」,「識別」,「監視」,「返済」の 4 つの連続する活動と，それら全体を通して必要となる「文書化」と「連絡」の 2 つの活動にまとめている．以下ではこれらの 8 種類を概説する．なお本論文では，文献 [49] のグループ化の考えに基づき,「計測」と「順位付け」を「監視」の小分類として扱う．

Prevention (防止) 累積負債や不注意による負債の発生を未然に防ぐ．

Identification (識別) ソフトウェアシステム内に存在する技術的負債を特定する．コードに対しては静的解析手法など，対象となる成果物に応じた手法を用いる．

図 2　技術的負債の管理に必要な 8 種類の活動

Monitoring (監視) 存在する技術的負債に対して，その影響の変化を監視する．利息の発生度合いを計測し，その値を可視化する手法などが採用される．監視では単に変化を追跡するだけでなく，返済の必要性を検討できるよう負債の深刻度に応じた順位付けを行う．

Measurement (計測) 負債を返済するために必要となるコストの推定手法を適用することで，識別された技術的負債の利益と利息を定量化する．

Prioritization (順位付け) 識別された技術的負債に対して，対応すべき負債の順位付けを行う．優先順位付けに関しては，順位付けの基準に関しては Ribeiro らのサーベイ論文 [48] に，返済に必要な労力の見積もりに基づく順位付けに関しては Alfayez らのサーベイ論文 [1] に詳しくまとめられている．また，Lenarduzzi らの最新のサーベイ論文 [34] では，順位付けに関しては様々な提案が乱立しており統一した見解がまだ存在せず，さらなる研究が必要であることが述べられている．

Repayment (返済) 技術的負債が発生している成果物に対して，後回しにしていた対応を実施し利息を含めて負債を軽減させる．例えば，リファクタリング [23] やリエンジニアリングなどの技術を用いてプログラムの内的構造を変更し，保守性を下げるソースコードの構造的な問題を取り除く．Java プロジェクトにおいて返済活動がどのように発生したかに関する実証実験 [15] や，Python プロジェクトにおいて同様の調査を行い Java との比較を行った実証的研究 [64] がある．

Documentation (文書化) プロジェクトの利害

関係者が持つ負債に対する懸念に対処するために，統一された方法で技術的負債の内容，発生箇所，負債の受け入れを判断した開発者などの情報を表現し，識別した技術的負債を体系的に管理する．

Communication（連絡） プロジェクトの利害関係者が現在の負債の状況を容易に理解できるように支援を行う．例えば，ダッシュボードや，技術的負債をプロジェクトの残作業 (backlog) に含めて表示する機能などを用いる．

技術的負債の管理に関する研究手法の多くはこれらの活動を部分的に支援するものである．新しい種類の技術的負債の定義と識別方法の提案や，識別ツールを用いて開発データを分析することで技術的負債の識別と計測を行い性質を調査する実証的研究が数多くなされてきた．

技術的負債の管理に関する活動全体を扱う研究として，Ampatzoglou らが技術的負債に対する取り組みを経済活動とみなして分析をしている[5]．また，Fernandes らの系統的マッピングによる論点の整理[19] では，技術的負債管理を行う上で検討すべき要素に関して 2016 年までの既存研究での議論を分類している．検討すべき要素は最終的に技術的負債に関わる意思決定に必要な要因，返済に必要なコストの見積もり手法，意思決定を行う手法の 3 つのグループに分かれ，合計で 12 個の要素としてまとめられている．

3.2 支援ツール

Rios らのサーベイ論文[49] には，技術的負債の管理に関する研究手法とツールがまとめられている．特に技術的負債の検出については，ツール支援の状況が Khomyakov らによる系統的レビュー[30] に詳しく説明がなされている．

Avgeriou らは代表的な 9 つのツールに関する機能や利用例などの比較を行っている[6]．この調査結果によれば，Cast[14] と静的解析ツールプラットフォームの SonarQube[†3] の TD プラグイン[35] が論文で言及されている回数が多く，技術的負債研究者が利用していることがうかがえる．これらのツールには技術的負債に対する返済コストを見積もる機能が提供されている．

また SonarQube だけでなく，NDepend の技術的負債検出機能[†4] についても技術者向け QA サイトの StackOverflow 上で他のツールより多く議論がなされており，技術的負債の管理に関心がある技術者に利用されていることがうかがえる．技術的負債に関する研究だけでなく，実際の開発における技術的負債の管理に実用的なレベルでツール支援が行われている．

4 類似概念との比較

ここまでで，ソフトウェア進化阻害要因の一概念として技術的負債を概観したが，ソフトウェア開発に長く携わった読者の中には，既視感を抱いた方も多いのではないだろうか．技術的負債の悪影響を議論するうえでは，負債の返済時に必要な追加作業コスト (利息) の影響が重要となる．2.2 節で述べた分類のいずれの種類の負債においても，利息は既存の実装の修正コストや不足機能の実現を阻害する既存実装箇所の修正コストを指す．このような，継続的なソフトウェア開発 (ソフトウェア進化) を阻害する要因となるソフトウェアやソフトウェア開発プロジェクト・環境の状態については，技術的負債として活発に議論される以前から，類似概念として表現され，類似技術の研究が行われてきた．本節では，そのようなソフトウェア進化阻害要因を扱う類似技術と技術的負債との関係を説明する．

4.1 アンチパターン

アンチパターン (anti-patterns)[9] は，Brown らによって 1998 年に提案されたソフトウェアパターンの一種であり，再利用可能な良い設計構造を表現するデザインパターンのコンセプトとは逆に，否定的な解法に対してソフトウェアパターンの考え方を適用しこれらをカタログ化したものである．アンチパターンは，主にソフトウェア設計の比較的大きな規模の欠陥を示しており，継続的なソフトウェア進化を阻害する一種

†3 https://www.sonarqube.org/

†4 https://www.ndepend.com/docs/technical-debt

の技術的負債とみなすことができる．

書籍[9]には，デザインパターンと同様のクラス設計レベルのパターン，アーキテクチャ構造に関するパターン，プロジェクト管理に関するパターンが含まれている．個々のアンチパターンを記述するテンプレートには，何が原因でそのような否定的な解法が選択されるのか(Typical Causes)，その結果として現れる特徴は何か(Symptoms and Consequences)といった避けるべき状況に関する説明と，それを改善するためのリファクタリング方針(Refactored Solution)が含まれている．

有名なアンチパターンに，The Blob や Stovepipe System などがある．The Blob (God Class ともいう)は，一つのクラスに責務が集中し保守性や移植性を低下させている状況を指したパターンであり，後述する不吉な臭いのひとつとしても取り上げられている．Stovepipe System は，全体的な思想が欠落したアーキテクチャ設計を示すパターンであり，場当たり的な拡張や他システムとの統合の結果として発生するとされている．

4.2 不吉な臭い

不吉な臭い(smell; 以下，臭い)[23][56]は，主にソースコードをはじめとするソフトウェア成果物に対する，後述するリファクタリングによる改善を必要とするような，保守性の低下要因となる兆候を指す．特に，Beck と Fowler によりリファクタリングの書籍[23]で紹介された，コードの不吉な臭い(bad smells in code)の概念は実務者にも広く知られている．明確に誤りであるとはいえないものの，改善を促すような兆候(symptom)を指しており，「臭い」という，善悪をはっきりさせない曖昧な表現をとっている．

アンチパターンが指すような悪設計も，リファクタリングのような改善が行われることが好ましいため，臭いの概念はアンチパターンとも類似している．ただし，個々の研究が従っている定義に依るものの，概ね以下のような違いがあると考えている．

- アンチパターンはソフトウェアパターンの作法に従って半形式的に記述される必要があるのに対し，臭いはその表現方法がまちまちであり，厳密性の低い表現に留まることも多い．もっとも，パターンカタログとして臭いを記述した試みはある[69]．
- 臭いのほうが，対象のより直接的な兆候を扱いがちであり，自動的な検出により親和性がある．このため，数多くの臭いの検出ツールが公開されている[21][25][41][67]．こういった検出ツールの多くはソースコードの静的解析に基づくため，静的解析ツールの出力に関する調査[29][71]と臭いも強い関係がある．
- 臭いの調査研究や検出手法のなかには，臭いをその有無として二分するだけではなく，臭いがあった場合にはその度合いを含めて扱うものがある．例えば，臭いの度合いを 1–10 の整数値で定めた深刻度(smell severity)[25][38]や，臭いの評価値として定義された強度(smell intensity)[20][21]がある．また，こういった考えを発展させ，臭いとして認識されるより前の段階で，臭いに近づいている状況を特定するために，腐敗度として 1 未満の深刻度を定義する試み[53]もある．

前述したように，アンチパターンとしても臭いとしても記述されているような対象もあり，これらの境界は明確ではない．

臭いは，対象のソフトウェア成果物や状況において累積負債に関係する利息が増大化していることの兆候や，不注意により意図せず生み出した負債が存在していることの兆候とみなせる．この考えに従い，臭いに関する活動も，技術的負債に関する活動に対応付く．例えば，コードの臭いに基づいてリファクタリングの適用を検討する場合において，臭いの検出と測定は負債の識別とその影響の計測に，リファクタリングの適用は負債の返済に対応付く．

技術的負債に関する分析や議論のいくつかは，こういった臭いに関する分析や議論としてもなされてきた．例えば，複数のサーベイ論文[1][34]による調査からもわかるように，前節で述べた管理活動の一つである負債の優先順位付けについての多数の試みがある．一方で，こういった優先順位付けを臭いや静的解析ツールの結果に対して行っているものもある[51][52][68]．負債に関する調査の中には，主にソフト

ウェア進化阻害要因を負債として扱う研究に注目し，必ずしもこういった類似概念に関する研究が網羅的にカバーされていないものもある点に注意が必要である．

ただし，技術的負債の管理の観点の進展と比較すると，臭いを直接の管理の対象とみなす考えは一般的ではない．重複コードに関するものなど，特定の臭いについてはしばしば管理の対象として扱われていたり[16][42][74]，自動的に検出された臭いや静的解析ツールの警告の追跡の試みがあったり[3][73]はするものの，開発速度等のトレードオフに基づき負債を借り入れ，それらを管理していくという考えは希薄である．Lahtiらの報告では，臭いやアンチパターンと負債との関係の分析，また臭いやアンチパターンの検出を通じての負債管理の実践について述べている[32]．

4.3 リファクタリング

3節で述べたように，技術的負債の返済にはリファクタリングが有効である．リファクタリングは，対象のソフトウェアが外部へ提供する機能を変更することなくその内部構造を変更する作業であり[23][45]，ソースコードに対するリファクタリングのカタログ[23]が有名である．

前述の通り，リファクタリングは技術的負債の返済行為ととらえられる．例えば，複数のクラスに分割して実装すべきところを一つのクラスにまとめて実装してしまった場合，のちの開発においてそのクラスが持つ責務が多くなり保守性が下がることがある．ここでの技術的負債は初期段階でクラスを分割しなかったことであり，その結果生じるプログラム理解の困難さや，保守コストの増大が負債の利息となる．この負債の返済は，対象のクラスを初期段階で分割すべきだったクラス構造に変更するリファクタリングや，そのクラスから部分的に切り出せる機能を別クラスに移すリファクタリングに相当する．

2.2節で述べた累積負債に対しては，特に定期的な改善を行うことで，利息の発生を抑えることが可能である．一方で，戦術的負債や不注意な負債の場合には，負債を発生させた時点から重大な影響があり返済のタイミングでは大規模なリファクタリングが必要となる．このような違いは，リファクタリングの研究においても，Flossリファクタリングと Root-canalリファクタリングの2つに分類されて議論されている[43]．前者は機能追加等の変更の最中に行うリファクタリングを指し，リスクの小さな負債の蓄積を防止する．後者は通常の変更とは切り分けて行われるリファクタリングを指し，リスクの大きな負債も含めて計画的な返済を行う．また，Rajlichは，通常の変更の前後に行うリファクタリングをプレファクタリング，ポストファクタリングと定義している[47]．前者は変更を行いやすくするための活動，後者は変更による保守性低下の影響の軽減のための活動であり，これらにより継続的なソフトウェア進化で発生する負債の影響を抑制させる．

5 技術的負債の種類

前節までで説明したように，技術的負債とは開発過程で適切な対応を後回しにしたことで生まれる懸念・問題点である．それらは，ソフトウェアの進化阻害要因として開発プロセス全体に様々な形で現れる．研究者らは，負債の影響が現れるソフトウェア成果物や開発工程に応じて，技術的負債に細分化された名称をつけている．Riosらのサーベイ論文[49]では，Liらの初期の10分類[37]を拡張し，技術的負債を15種類にまとめている．また，Ernstらの書籍[18]でも一部同様の分類を含む8種類の分類が採用されている．これらの分類において技術的負債として認識されている進化阻害要因は，これまでソフトウェア工学の研究の対象となってきた問題が多く，4節で述べた類似概念で取り扱われた問題も少なくない．

以下では，代表的な名称とそれに関係する既存研究領域や周辺技術を紹介する．まず，特に研究事例の多い設計とコードに関する技術的負債について説明し，続いて他の種類についても概観する．

5.1 設計に関する技術的負債

設計に関する技術的負債は，研究者によって分類・定義が分かれる．多くのケースでは，ソフトウェアアーキテクチャレベルの設計に関連した技術的負債と局所的なモジュール構造上の技術的負債とを区別

する．

5.1.1 Design Debt

ソースコードの設計において妥協することで発生する負債を指す[37]．クラス間の局所的な関係といったモジュールの詳細な設計において発見される．後述するより抽象度の高い設計構造で発見されるArchitectural Debt と区別しない場合[18] もある．

典型的な具体例に，アンチパターンや不吉な臭いとして知られ God Class[9] などと呼ばれる大量のメソッドを保有し極端に肥大化したクラスや，密結合しているクラス群など機能独立性やオブジェクト指向設計の SOLID 原則[39] の観点から適していないクラス設計がある．God Class は，内包する機能に対する拡張をする際にその理解コストが高くなり保守性が低下する．また，密結合しているクラス群は変更の影響が結合するクラスに伝播するため影響解析と試験が必要となり開発コストが増大する．

このような問題は古くから設計手法や設計評価の研究で議論されており，CK メトリクス[11] や MOOD[17] などの設計メトリクスによる品質計測と評価[33] が広く用いられてきた．また 4 節で述べたコードの臭いやリファクタリングは，Design Debt の識別と返済に有効な既存技術である．

5.1.2 Architectural Debt

ソフトウェアのアーキテクチャ上に見られる負債であり，コードの簡単な変更では解決できない負債を指す．保守性など内部品質の観点での妥協が原因で発生する[37]．

開発が進む過程で，局所的な設計を優先した結果不適切な依存関係を含めてしまうなどの原因で発生する状況であり，結果的に保守確認のコストが増大する負債となる．典型的な具体例として，循環的な依存関係が挙げられる．パッケージ構造に関する設計原則 The Acyclic Dependencies Principle (ADP)[39] などでもまとめられているように，アーキテクチャを構成する部品間で循環的な依存関係を含むと，更新の影響が循環して波及し保守確認が困難となる．

このように，設計として選択したアーキテクチャを逸脱する変更によって負債が発生する状況は，前節で述べたアンチパターンに含まれているほか，以前から荒廃 (decay)，ずれ (drift) などの用語でその逸脱を表現し議論がなされている[50]．また，初期のアーキテクチャから逸脱するという考え方だけでなく，アーキテクチャとは設計判断 (design decision) の集合である[28] という考えのもと，その設計判断を復元する[55] という取り組みも行われている．Soliman らは実プロジェクトを分析し，技術的負債を発生させた設計判断の特徴を調査する実証的な研究を行っている[58]．

Besker らは Architectural Debt に特化したサーベイを行い Architectural Debt を識別するための特徴と，負債の影響に関する詳細な議論を行っている[8]．

5.2 コードに関する技術的負債

ソースコード内で発見される負債は Code Debt[37] と呼ばれる．これらは可読性や保守性に悪影響を与える記述を指し，例えば不適切な識別子名や Cyclomatic 複雑度の高いコード，デッドコードや重複コード (コードクローン) の存在，コーディング規約に違反する記述の存在を指す．Design Debt もソースコードを解析することで検出できる負債であることから，Ernst らは Implementation Debt としている[18]．コードに関する技術的負債も 4 節で述べた不吉な臭いと定義されているものが多く含まれている．例えば，Wake のカタログ[69] では，デッドコードは不必要な複雑性に関連する臭いに分類される “実行されないコード” として紹介されている．重複コードについても同様にクラス内の臭いの一つとして定義されている．

Code Debt の多くは対象ソフトウェアの実行を直接妨げることはないため，しばしば意図的に見逃される．しかしながら，それらは可読性や保守性に悪影響を与える．個々の Code Debt は小さな影響であっても，その存在を放置することが続くと 2.2 節で議論した累積負債となる．

Code Debt として扱われる既存研究に，コードクローン，コーディングルール違反などの低品質なコードに関する研究が挙げられる．

コードクローンについては技術的負債の考え方が導入される以前から研究が進んでおり，3 節で議論した活動のうち識別に相当する検出技術[75]，返済に相当するリファクタリング手法[76]，監視に相当する管

理支援技術[74]に関する解説論文が発表されている．

コーディングルール違反は，開発組織や業界団体，コミュニティが定めるコーディング規約や，経験的に知られているガイドラインなどを静的解析によって検出する静的コード検査の結果が用いられる．コーディング規約は，自動車業界が定めたMISRA Cガイドライン[26]やCERTが定めたCERT C Secure Coding Standard (CCSCS)のように誤りを混入しやすい記述を防止するもので，これらに特化した検出ツールの提案[44]や，コーディング規約違反が与える影響について研究[10][59]がなされている．また，経験的に知られている汎用的なコーディングルールに基づく静的コード検査については，商用ツールを含む解析基盤が用意されていることから広く用いられている．

一方で，静的解析に基づく検査ツールの示す警告は偽陽性が高いこと[12][29]が知られているため，大量に出力される警告に含まれる重要な警告に注目できるよう支援する必要がある．これまでに，以前から存在する警告は深刻度が低いか偽陽性と判断されたと仮定し，進化の過程で新規に出現した警告に着目できるよう過去の警告を追跡して管理する手法[73]が提案されている．また新規性以外の基準で警告箇所を順位付けする試みとして，開発者が警告をどのように取り扱っているかを調査し開発の状況(context)が重要であることを示した研究[68]やActionable Alert Identification Techniques (AAIT)と呼ばれる研究領域が存在し多くの手法が提案されている[27]．これらの手法では，警告の特徴，コードの特徴，プロジェクトリポジトリの特徴などを入力とし，機械学習やグラフ構造による重みづけを用いて優先付けを行う．

これらの検査警告をCode Debtとして扱い順位付けを行うには，警告が発生している箇所やその警告内容に応じた負債の影響を考慮して順位付けが行われるべきである．例えば，機能拡張等の変更が予定されていない箇所での警告はその他の箇所と比べて小さな負債ととらえることができる．しかしながら，静的コード検査の警告を負債として順位付けを行う研究は少ない．Code Debtの優先順位付けとして利用できる手法に，OASIS[71]がある．Android Appsを対象にするこの手法では，App StoreのUserレビューの文章を学習時に用いることで，セキュリティやパフォーマンスなど非機能要求に影響を与える静的コード検査警告に重みづけを行いPrecisionを上げることに成功している．

5.3 その他の種類

これまでの技術的負債の研究の多くは前述の設計とコードに関する技術的負債であるが，その他の成果物や工程に起因するソフトウェア進化の阻害要因を技術的負債の考え方で説明する提案もなされている．以下ではその代表的なものがどのような観点を負債としてとらえているかを簡単に説明する．

- Test Debt[18][37]
  ソフトウェアテストの品質を下げるものであり，結果としてソフトウェアの品質保証に悪影響を与える事象．例えば，カバレッジが低いテストスイートや実行されない不必要なテストケース，開発者の設定したテストケースが少ないことを指す．
- Documentation Debt[18][37]
  ソフトウェア成果物内に記載すべき文書情報に発生する負債．例えば，必要なコードコメントや情報が欠損している．更新不足・不適切・不完全な記述が含まれる場合などを指す．
- Defect Debt[37]
  修正すべきだが後回しになっている既知のバグを指す負債．
- Infrastructure Debt[37]
  プロジェクトの進行を妨げるインフラ上の懸念事項．例えば，古い技術の利用，継続的インテグレーションや開発環境で構築したソフトウェアを実行環境へ配備する作業を自動化していないことなどを指す．Ernstらの書籍[18]では，Deployment Debtとしている．
- Requirements Debt[18][37]
  要求仕様書で発見される負債．矛盾する要求記述や部分的にしか実現されない機能要求や非機能要求が存在することなど挙げられる．
- People Debt[49], Process Debt[49]

Social Debt [18] とも呼ばれるものであり，開発チーム内でのメンバー間の関係や開発プロセスで発見される負債である．例えば，メンバーの関係と開発担当範囲の関係が大きく異なる場合，開発の際に必要となる情報共有が不十分となり結果的に開発の効率が悪くなることが知られている [60] [61]．これはコミュニティの臭い (community smell) [62] とも呼ばれる．

- Build Debt [37]
  不要なコードのビルド，誤った依存関係など，ビルド方法に起因する負債であり，Make や Apache Ant などの構築管理ツールや Apache Maven, Gradle などの構成管理と構築管理を実現するツールの設定ファイルで発見される．
- Versioning Debt [37]
  不必要な fork，複数リリースラインに対する保守対応といった，対象ソフトの版管理構造に起因する負債．

また，成果物や工程の観点ではなく，特定の問題ドメインに固有の技術的負債も議論されている．Ernst らの書籍 [18] では，機械学習システムに固有の技術的負債 [54] についても議論がなされている．

6 おわりに

本論文では，多くのサーベイ論文が存在し活発に議論がなされている技術的負債に関して，その定義およびその周辺技術について概観し，これまでの技術とどのような関係にあるのかを解説した．4 節で述べたように，技術的負債はソフトウェア進化の阻害要因として認識されてきた既存概念やその対応策と本質的な差異はない．技術的負債の考え方では，ソフトウェア進化に悪影響を与え得る状況を識別して監視し，過度に悪影響を与える前にリファクタリング等で返済するという管理プロセスが新しい捉え方である．

顧客のニーズや利用環境の変化に迅速に対応する必要がある現在のソフトウェア開発においては，技術的負債を負うことは避けられない選択である．どのような技術的負債であれば問題ないのか，技術的負債の管理の適切や方法や必要となる支援は何かという観点は今後益々重要になると考える．

技術的負債の観測や順位付けについては，開発者がどのように管理しているか不明な点がまだ多い．実際に技術的負債がどのように発生し返済されていったかの分析を可能にするためには，2.3 節で述べた SATD のように，その発生から返済まで過去の開発履歴から観測可能な技術的負債について情報収集し，実証的な研究を行うことで知見を得ることが有効であると考える．

また，これまでに多くの研究がなされてきた設計や実装に関する技術的負債であっても，負債の発生や存在の全てを機械的に検出することがいまだ困難である．より複雑な技術的負債の検出を可能とする探索的なアプローチや，不注意による負債の混入を防止するために技術的負債を類型化してパターン化する取り組みが今後期待される．

謝辞 本論文の執筆にあたり，JSPS 二国間交流事業 Joint International Seminar on Technical Debt (JPJSBP120214402) での議論が参考となった．ここに感謝の意を表す．

参考文献

[1] Alfayez, R., Alwehaibi, W., Winn, R., Venson, E., and Boehm, B.: A Systematic Literature Review of Technical Debt Prioritization, *Proc. International Conference on Technical Debt (TechDebt)*, 2020, pp. 1–10.

[2] Allman, E.: Managing Technical Debt, *Communications of the ACM*, Vol. 55, No. 5(2012), pp. 50–55.

[3] Aloraini, B., Nagappan, M., German, D. M., Hayashi, S., and Higo, Y.: An Empirical Study of Security Warnings from Static Application Security Testing Tools, *Journal of Systems and Software*, Vol. 158(2019), pp. 1–25.

[4] Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F., and Seaman, C.: Identification and management of technical debt: A systematic mapping study, *Information and Software Technology*, Vol. 70(2016), pp. 100–121.

[5] Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., and Avgeriou, P.: The financial aspect of managing technical debt: A systematic literature review, *Information and Software Technology*, Vol. 64(2015), pp. 52–73.

[6] Avgeriou, P. C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D. D., de Toledo, S. S.,

and Tsintzira, A. A.: An Overview and Comparison of Technical Debt Measurement Tools, *IEEE Software*, Vol. 38, No. 3(2021), pp. 61–71.

[7] Behutiye, W. N., Rodriguez, P., Oivo, M., and Tosun, A.: Analyzing the concept of technical debt in the context of agile software development: A systematic literature review, *Information and Software Technology*, Vol. 82(2017), pp. 139–158.

[8] Besker, T., Martini, A., and Bosch, J.: Managing architectural technical debt: A unified model and systematic literature review, *Journal of Systems and Software*, Vol. 135(2018), pp. 1–16.

[9] Brown, W. J., Malveau, R. C., McCormick, H. W., and Mowbray, T. J.: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley and Sons, 1998.

[10] Cathal Boogerd, L. M.: Assessing the value of coding standards: An empirical study, *Proc. International Conference on Software Maintenance (ICSM)*, pp. 277–286.

[11] Chidamber, S. R. and Kemerer., C. F.: A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6(1994), pp. 476–493.

[12] Christakis, M. and Bird, C.: What developers want and need from program analysis: an empirical study, *Proc. International Conference on Automated Software Engineering (ASE)*, 2016, pp. 332–343.

[13] Cunningham, W.: The WyCash Portfolio Management System, OOPSLA '92 Experience Report, 1992. `http://c2.com/doc/oopsla92.html` (2022/11/1 accessed).

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

[15] Digkas, G., Lungu, M., Avgeriou, P., Chatzigeorgiou, A., and Ampatzoglou, A.: How do developers fix issues and pay back technical debt in the Apache ecosystem?, *Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 153–163.

[16] Duala-Ekoko, E. and Robillard, M. P.: Tracking Code Clones in Evolving Software, *Proc. International Conference on Software Engineering (ICSE)*, 2007, pp. 158–167.

[17] e Abreu, F. B., Goulão, M., and Esteve, R.: Toward the Design Quality Evaluation of Object-Oriented Software Systems, *Proc. International Conference on Software Quality*, 1995.

[18] Ernst, N., Kazman, R., and Delange, J.: *Technical Debt in Practice: How to Find It and Fix It*, MIT Press, 2021.

[19] Fernández-Sánchez, C., Garbajosa, J., Yagüe, A., and Perez, J.: Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study, *Journal of Systems and Software*, Vol. 124(2017), pp. 22–38.

[20] Fontana, F. A., Ferme, V., and Zanoni, M.: Poster: Filtering Code Smells Detection Results, *Proc. International Conference on Software Engineering (ICSE)*, 2015, pp. 803–804.

[21] Fontana, F. A., Ferme, V., Zanoni, M., and Roveda, R.: Towards a Prioritization of Code Debt: A Code Smell Intensity Index, *Proc. International Workshop on Managing Technical Debt (MTD)*, 2015, pp. 16–24.

[22] Fowler, M.: TechnicalDebtQuadrant, 2009. `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html` (2022/11/1 accessed).

[23] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[24] Fucci, G., Zampetti, F., Serebrenik, A., and Di Penta, M.: Who (Self) Admits Technical Debt?, *Proc. International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 672–676.

[25] Ganea, G., Verebi, I., and Marinescu, R.: Continuous Quality Assessment with inCode, *Science of Computer Programming*, Vol. 134(2017), pp. 19–36.

[26] Hatton, L.: Safer language subsets: an overview and a case history, MISRA C, *Information and Software Technology*, Vol. 46, No. 7(2004), pp. 465–472.

[27] Heckman, S. and Williams, L.: A systematic literature review of actionable alert identification techniques for automated static code analysis, *Information and Software Technology*, Vol. 53, No. 4(2011), pp. 363–387.

[28] Jansen, A. and Bosch, J.: Software Architecture as a Set of Architectural Design Decisions, *Proc. Working Conference on Software Architecture (WICSA)*, 2005, pp. 109–120.

[29] Johnson, B. et al.: Why don't software developers use static analysis tools to find bugs?, *Proc. International Conference on Software Engineering (ICSE)*, 2013, pp. 672–681.

[30] Khomyakov, I., Makhmutov, Z., Mirgalimova, R., and Sillitti, A.: Automated Measurement of Technical Debt: A Systematic Literature Review, *Proc. International Conference on Enterprise Information Systems (ICEIS)*, 2019, pp. 95–106.

[31] Kruchten, P., Nord, R. L., and Ozkaya, I.: Technical Debt: From Metaphor to Theory and Practice, *IEEE Software*, Vol. 29, No. 6(2012), pp. 18–21.

[32] Lahti, J. R., Tuovinen, A.-P., and Mikkonen, T.: Experiences on Managing Technical Debt with Code Smells and AntiPatterns, *Proc. International Conference on Technical Debt (TechDebt)*, 2021, pp. 36–44.

[33] Lanza, M. and Marinescu, R.: *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, Springer, 2006.

[34] Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., and Arcelli Fontana, F.: A systematic literature

review on Technical Debt prioritization: Strategies, processes, factors, and tools, *Journal of Systems and Software*, Vol. 171(2021), pp. 110827:1–16.

[35] Letouzey, J.-L.: The SQALE method for evaluating Technical Debt, *Proc. International Workshop on Managing Technical Debt (MTD)*, 2012, pp. 31–36.

[36] Li, Y., Soliman, M., and Avgeriou, P.: Automatic Identification of Self-Admitted Technical Debt from Different Sources, *Empirical Software Engineering*, Vol. 28, No. 65(2023), pp. 1–38.

[37] Li, Z., Avgeriou, P., and Liang, P.: A systematic mapping study on technical debt and its management, *Journal of Systems and Software*, Vol. 101(2015), pp. 193–220.

[38] Marinescu, R.: Assessing Technical Debt by Identifying Design Flaws in Software Systems, *IBM Journal of Research and Development*, Vol. 56, No. 5(2012), pp. 9:1–13.

[39] Martin, R. C.: アジャイルソフトウェア開発の奥義 第 2 版オブジェクト指向開発の神髄と匠の技, SB クリエイティブ, 2008.

[40] McConnell, S.: Managing Technical Debt, Technical report, Construx Software, 2008.

[41] Moha, N., Guéhéneuc, Y.-G., Duchien, L., and Meur, A.-F. L.: DECOR: A Method for the Specification and Detection of Code and Design Smells, *IEEE Transactions on Software Engineering*, Vol. 36, No. 1(2010), pp. 20–36.

[42] Mondal, M., Roy, C. K., and Schneider, K. A.: A survey on clone refactoring and tracking, *Journal of Systems and Software*, Vol. 159(2020), pp. 110429:1–27.

[43] Murphy-Hill, E. and Black, A. P.: Refactoring Tools: Fitness for Purpose, *IEEE Software*, Vol. 25, No. 5(2008), pp. 38–44.

[44] Olesen, M. C., Hansen, R. R., Lawall, J. L., and Palix, N.: Coccinelle: Tool support for automated CERT C Secure Coding Standard certification, *Science of Computer Programming*, Vol. 91(2014), pp. 141–160.

[45] Opdyke, W. F.: *Refactoring Object-Oriented Frameworks*, PhD Thesis, University of Illinois at Urbana-Champaign, 1992.

[46] Potdar, A. and Shihab, E.: An Exploratory Study on Self-Admitted Technical Debt, *Proc. International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 91–100.

[47] Rajlich, V.: *Software Engineering: The Current Practice*, Chapman and Hall – CRC, 2011.

[48] Ribeiro, L. F., Farias, M. A. d. F., Mendonça, M., and Spínola, R. O.: Decision Criteria for the Payment of Technical Debt in Software Projects: A Systematic Mapping Study, *Proc. International Conference on Enterprise Information Systems (ICEIS)*, 2016, pp. 572–579.

[49] Rios, N., de Mendonça Neto, M. G., and Spínola, R. O.: A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners, *Information and Software Technology*, Vol. 102(2018), pp. 117–145.

[50] Rosik, J., Le Gear, A., Buckley, J., Babar, M. A., and Connolly, D.: Assessing architectural drift in commercial software development: a case study, *Software: Practice and Experience*, Vol. 41, No. 1(2011), pp. 63–86.

[51] Sae-Lim, N., Hayashi, S., and Saeki, M.: How Do Developers Select and Prioritize Code Smells? A Preliminary Study, *Proc. International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 484–488.

[52] Sae-Lim, N., Hayashi, S., and Saeki, M.: Context-Based Approach to Prioritize Code Smells for Prefactoring, *Journal of Software: Evolution and Process*, Vol. 30, No. 6(2018), pp. e1886:1–24.

[53] Sae-Lim, N., Hayashi, S., and Saeki, M.: Supporting Proactive Refactoring: An Exploratory Study on Decaying Modules and Their Prediction, *IEICE Transactions on Information and Systems*, Vol. E104-D, No. 10(2021), pp. 1601–1615.

[54] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D.: Hidden Technical Debt in Machine Learning Systems, *Advances in Neural Information Processing Systems*, Vol. 28 (NIPS 2015), 2015.

[55] Shahbazian, A., Kyu Lee, Y., Le, D., Brun, Y., and Medvidovic, N.: Recovering Architectural Design Decisions, *Proc. International Conference on Software Architecture (ICSA)*, 2018, pp. 95–104.

[56] Sharma, T. and Spinellis, D.: A survey on software smells, *Journal of Systems and Software*, Vol. 138(2018), pp. 158–173.

[57] Sierra, G., Shihab, E., and Kamei, Y.: A survey of self-admitted technical debt, *Journal of Systems and Software*, Vol. 152(2019), pp. 70–82.

[58] Soliman, M., Avgeriou, P., and Li, Y.: Architectural design decisions that incur technical debt – An industrial case study, *Information and Software Technology*, Vol. 139(2021), pp. 106669:1–17.

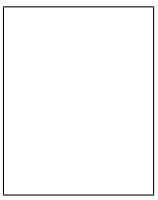
[59] Takai, Y., Kobayashi, T., and Agusa, K.: Software Metrics Based on Coding Standards Violations, *Proc. Joint Conference of International Workshop on Software Measurement and International Conference on Software Process and Product Measurement (IWSM-MENSURA)*, 2011, pp. 273–278.

[60] Tamburri, D. A.: Software Architecture Social Debt: Managing the Incommunicability Factor, *IEEE Transactions on Computational Social Systems*, Vol. 6, No. 1(2019), pp. 20–37.

[61] Tamburri, D. A., Kruchten, P., Lago, P., and van Vliet, H.: Social debt in software engineering: insights from industry, *Journal of Internet Services and Applications*, Vol. 6, No. 1(2015), pp. 10:1–17.

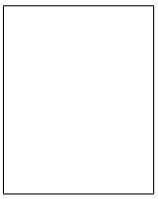
[62] Tamburri, D. A., Palomba, F., and Kazman, R.:

Exploring Community Smells in Open-Source: An Automated Approach, *IEEE Transactions on Software Engineering*, Vol. 47, No. 3(2021), pp. 630–652.
[63] Tan, J., Feitosa, D., and Avgeriou, P.: Do practitioners intentionally self-fix Technical Debt and why?, *Proc. International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 251–262.
[64] Tan, J., Feitosa, D., Avgeriou, P., and Lungu, M.: Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem, *Journal of Software: Evolution and Process*, Vol. 33, No. 4(2021), pp. e2319:1–25.
[65] Tang, A. and Kazman, R.: Decision-Making Principles for Better Software Design Decisions, *IEEE Software*, Vol. 38, No. 6(2021), pp. 98–102.
[66] Tom, E., Aurum, A., and Vidgen, R.: An exploration of technical debt, *Journal of Systems and Software*, Vol. 86, No. 6(2013), pp. 1498–1516.
[67] Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A.: JDeodorant: Identification and Removal of Type-checking Bad Smells, *Proc. European Conference on Software Maintenance and Reengineering (CSMR)*, 2008, pp. 329–331.
[68] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., and Gall, H. C.: Context is king: The developer perspective on the usage of static analysis tools, *Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2008, pp. 329–331.
[69] Wake, W. C.: *Refactoring Workbook*, Addison-Wesley, 2003.
[70] Wattanakriengkrai, S., Maipradit, R., Hata, H., Choetkiertikul, M., Sunetnanta, T., and Matsumoto, K.: Identifying Design and Requirement Self-Admitted Technical Debt Using N-gram IDF, *Proc. 9th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, 2018, pp. 7–12.
[71] Wei, L., Liu, Y., and Cheung, S.-C.: OASIS: Prioritizing Static Analysis Warnings for Android Apps Based on App User Reviews, *Proc. Joint Meeting on Foundations of Software Engineering (FSE)*, 2017, pp. 672–682.
[72] Zampetti, F., Noiseux, C., Antoniol, G., Khomh, F., and Di Penta, M.: Recommending when Design Technical Debt Should be Self-Admitted, *Proc. International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 216–226.
[73] 渥美紀寿, 桑原寛明: MAFP: ソースコードに対する静的検査における警告の管理ツール, コンピュータソフトウェア, Vol. 33, No. 4(2016), pp. 50–66.
[74] 堀田圭佑, 肥後芳樹, 楠本真二: 生成抑止，分析効率化，不具合検出を中心としたコードクローン管理支援技術に関する研究動向, コンピュータソフトウェア, Vol. 31, No. 1(2014), pp. 14–29.
[75] 神谷年洋, 肥後芳樹, 吉田則裕: コードクローン検出技術の展開, コンピュータソフトウェア, Vol. 28, No. 3(2011), pp. 29–42.
[76] 肥後芳樹, 吉田則裕: コードクローンを対象としたリファクタリング, コンピュータソフトウェア, Vol. 28, No. 4(2011), pp. 43–56.



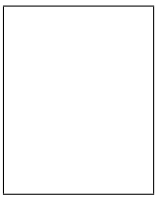
小林 隆志

1999 年東京工業大学大学院情報理工学研究科修士課程修了．2002 年同大学助手．名古屋大学特任助教授，准教授，東京工業大学准教授を経て，2021 年より東京工業大学情報理工学院教授．博士 (工学)．ソフトウェア再利用技術，リポジトリマイニング，プログラム理解などの研究に従事．



林 晋平

2008 年東京工業大学大学院情報理工学研究科計算工学専攻博士後期課程修了．同専攻助教を経て，2018 年より同大学情報理工学院准教授．博士 (工学)．ソフトウェア進化やソフトウェア開発環境の研究に従事．



斎藤 忍

2001 年慶應義塾大学大学院修士課程修了，同年株式会社 NTT データに入社．2015 年日本電信電話株式会社に転籍．現在，コンピュータ&データサイエンス研究所に所属．2016〜2018 年カリフォルニア大学アーバイン校客員研究員．博士（工学）．ソフトウェア工学，要求工学に関する研究開発に従事．