情報処理学会論文誌 Vol.61 No.4 908–917 (Apr. 2020)

リポジトリマイニングに基づくアンチパターン検出手法

市井 誠[1,a)] 川上 真澄[1]

受付日 2019年7月31日, 採録日 2020年1月16日

概要：長年の開発により大規模化したソフトウェアに対する，効果的なリファクタリングを支援するためのアンチパターン検出手法を提案する．提案手法は，リポジトリマイニングを用いたロジカルカップリング分析に基づき，変更の分散（*Shotgun Surgery*），肥満児（*The Blob*），重複したコード（*Duplicated Code*）の 3 種類のアンチパターンを検出する．提案手法の評価のため，C 言語および C++言語で記述された，大規模なオープンソースソフトウェア（OSS）および組込みシステム製品のソフトウェアを用いた適用実験を行った．組込システム製品の開発者ヒアリングにより，開発者の問題意識に適合したアンチパターンを検出できることを確認した．

キーワード：リファクタリング，アンチパターン，リポジトリマイニング，ロジカルカップリング

Anti-pattern Detection Based on Repository Mining

Makoto Ichii[1,a)] Masumi Kawakami[1]

Received: July 31, 2019, Accepted: January 16, 2020

Abstract: We propose an anti-pattern detection approach based on software repository mining for supporting effective refactoring against long-lived, large-scale software systems. Our method detects *Shotgun Surgery*, *The Blob* and *Duplicated Code* using logical coupling analysis based on repository mining. As a case study, we detect anti-patterns from an open-source software system and industrial embedded software systems written in C or C++ language using our method. In addition, we had an interview with the developers of the embedded software systems about the detected anti-pattern instances. The result indicates that our method successfully detects the anti-pattern instances where the developers consider that some design issues exist.

Keywords: refactoring, anti-pattern, repository mining, logical coupling

1. はじめに

大規模な組込みソフトウェア開発では，同じコードベースを 10 年以上にわたり利用し続けることも珍しくない．リファクタリングは，繰り返される修正にともなう技術的負債の蓄積を避け，開発効率を維持するために必要であるが，従来，変更影響範囲を増やし，不具合リスクを生む余計な変更として避けられてきた．特に組込みソフトウェアにおいては，変更時の検証コストの大きさから，製品品質の維持のため変更の局所化が優先されてきた．しかし，蓄積した技術的負債は工程を圧迫し，いずれ品質にも影響するため，リファクタリングは避けて通れない道であるといえる．

一方で，大規模化したソフトウェアに対し，無闇にリファクタリングを行っても，変更および検証のコストがかさむだけで，将来の開発効率や品質の向上につながらないおそれがある．そのため，悪影響を生む箇所を的確に抽出し，適切な方法でリファクタリングすることが必要である．しかし，そのためには製品開発状況の把握と，リファクタリングに関する習熟が必要となり，熟練者以外には難しい．

この課題に対し，著者らは，リファクタリングすべき箇所およびその方法を開発者に推薦することをめざした取組みを進めている．本論文では，リポジトリマイニングに

[1] 株式会社日立製作所研究開発グループシステムイノベーションセンタシステム生産性研究部
System Productivity Research Department Center of Technology Innovation - Systems Engineering Research & Development Group, Hitachi, Ltd., Totsuka, Yokohama, Kanagawa 244–0817, Japan
[a)] makoto.ichii.dn@hitachi.com

より，ソフトウェアのリファクタリングすべき箇所をアンチパターンとして自動的に検出する手法を提案する．提案手法は，組込みソフトウェアのソースコードへの適用を念頭におき，変更の分散（*Shotgun Surgery*），肥満児（*The Blob*），重複したコード（*Duplicated Code*）の 3 種類のアンチパターンを，ロジカルカップリング分析とプログラム解析の組合せにより検出する．

以下，2 章にてリファクタリングすべき箇所の検出に関する関連研究を示したうえで，3 章にて提案するアンチパターン検出手法について述べる．4 章に提案手法の適用実験について述べ，5 章にて適用実験の結果に関する考察を述べる．最後に 6 章にてまとめと今後の課題について述べる．

2. リファクタリングすべき箇所の検出

Mens らは，リファクタリングのステップを以下のとおりに区分している [1]．1. ソフトウェアの中からリファクタリングすべき箇所を特定する．2. 適用すべきリファクタリングを決定する．3. リファクタリングにより振舞いが保持されることを保証する．4. ソースコードにリファクタリングを適用する．5. リファクタリングの効果を測定する．6. ドキュメントなどの他の成果物との整合性をとる．本研究はこの流れのなかでの 1. を主に対象とし，さらに 2. に対する指針を与えることを目的とする．

リファクタリングすべき箇所を特定するための単純な方法として，規模や複雑さなど，品質に関するメトリクス値を用い，ある閾値をこえる要素を候補とする方法がある．品質にかかわるメトリクス値としては，行数（LOC）および循環的複雑度，CK メトリクス [2] などが存在する．この方法は，実際のソフトウェアに適用すると膨大な候補が検出されることに加え，特定メトリクス値を向上させるリファクタリングが必ずしも良い設計とはいえず，適切なリファクタリングにつながりにくい問題点がある．Ó Cinnéide らは，メトリクス値とリファクタリングの効果との間の明確なつながりは存在しないと結論付けている [3]．

Fowler の文献 [4] では，76 種のリファクタリングを示すとともに，リファクタリングすべき箇所を示す不吉な兆候として，22 種類の「不吉な匂い（Bad smell）」を説明している．不吉な匂いは「アンチパターン」とも呼ばれる．不吉な匂いの例としては，「重複したコード（Duplicated Code）」や「長すぎるメソッド（Long Method）」がある．また，Kerievsky の文献 [5] においても，同様の概念として 12 種類の「コードの匂い（code smell）」を取り上げている．これらの概念は，設計上の問題点を示すと同時に適用すべきリファクタリングを検討する指針として機能する．たとえば，Fowler の文献では「長すぎるメソッド」への対処方法として「メソッドの抽出」や「問合せによる一時変数の置き換え」など 4 種類の方法があげられている．ただし，開発者が感覚を養うべきとの立場から，それらの定義は定性的に示されるのみであり，自動的な検出については言及されていない [4]．本研究はアンチパターンの自動検出を目的とする．本章の残りの部分にて，アンチパターンの自動検出に関する既存研究について述べたうえで，本研究の位置づけについて述べる．

アンチパターンの自動的な特定にはソースコード解析に基づく手法やリポジトリマイニングに基づく手法など様々な手法が提案されている．CCFinder などのコードクローン検出ツールは，「重複したコード」を自動的に検出するツールである [6], [7]．重複したコードは不吉な匂いのなかでも開発者に問題点として意識されやすいものであり [3], [8]，改善の方法も「メソッドの抽出」など比較的検討が容易なものとなる．一方，ツールは機械的に重複/類似コードを検出するため，検出結果が必ずしも改善すべき箇所とは限らない [9]．たとえば意図的/偶然の重複や，言語仕様上の制約に基づくイディオムなど，リファクタリングの対象とならないものも検出結果には含まれる．また，すべてのコードクローンが保守性に悪影響をもたらすとは限らないため，変更せず実害のないところは経過観察など，将来を見越した絞り込みが必要となる．

Moha らは，“Code Smell” を構造から計測したメトリクス値の組合せとして定義したうえで，複数のアンチパターンを Code Smell の組合せとして定義して検出するツール DÉCOR を提案している [10]．DÉCOR は，不吉な匂いやアンチパターンに関する定性的な記述を，定量化可能な指標の組合せで定義することを試みたものである．DÉCOR もコードクローン検出と同様，プログラム構造からの機械的な検出のため，リファクタリングに際しては適切な絞り込みが必要となる．

Palomba らは変更履歴を用いたロジカルカップリング分析の結果に基づき，Shotgun Surgery などのバッドスメルを検出するツール HIST を提案している [11]．ロジカルカップリング分析とは，同時にコミットされたプログラムエンティティ（ファイル，関数など）に対してアソシエーション分析を適用することで，同時に変更されやすい関係を抽出する技術である [12]．

本研究では，開発者が適用すべきリファクタリング方法を決定する指針を得られるよう，アンチパターンが顕在化している箇所をリファクタリングすべき箇所として検出する．また，基本的なアプローチとして HIST 同様にリポジトリマイニングに基づくロジカルカップリング分析を用いる．リポジトリマイニングに基づく方法は，開発履歴の中から実害の痕跡を検出していると見なすこともでき，歴史の長い大規模システムに向いていると考えられるためである．HIST は Java プログラムを対象として検出するアンチパターンを定めているが，本研究では，組込みソフトウェアにて多く利用される C 言語向けに，HIST では考慮して

いないプログラム解析ベースの特徴も用いた検出を行う．

3. アンチパターン検出手法

3.1 検出対象のアンチパターン

本論文では，長く開発されてきた大規模な組込みソフトウェアへの適用を念頭に，アンチパターン検出手法を提案する．検出するアンチパターンは，組込みソフトウェアではC言語が用いられることが多いことを考慮し，手続き指向で設計されたソフトウェアに適用可能なものとする．さらに，開発者の問題意識を喚起できるよう，構造だけでなく，実際に発生した良くない変更のパターンとして定義できるものに注目する．

上記を考慮し，本論文では以下のアンチパターンを検出対象とする．

- 変更の分散（Shotgun Surgery）：ある変更が他の多数の小さな変更を引き起こす．本論文では異なる関数の関係としてとらえる．
- 肥満児（The Blob）：ある要素が責務を抱え込み肥大化している．本論文では，責務の抱え込みにより，様々な変更についての変更影響を受けやすいという特徴に注目する．
- 重複したコード（Duplicated Code）：重複したコード辺が存在する．本論文では，異なる関数間にてコード辺が重複し，さらに同時に変更されるものに絞って考える．

変更時の影響に関するものや，肥大化・複雑化に関するものは開発者の関心も高い．本論文では特に関数間の関係に関するものに注目しているが，長すぎるメソッド（Long method）などの，関数内部の構造に関するアンチパターンについては，本論文とは異なるアプローチでの検出を検討している [13]．

3.2 リポジトリマイニングに基づくロジカルカップリング分析

ロジカルカップリング（Logical coupling）は，あるソフトウェア成果物の要素を変更するとき，別の要素もまた変更されるという関連のことを指す [12], [14], [15]．ここでソフトウェア成果物の要素は様々に定義される．たとえば関数やクラス，ソースファイルといったプログラム要素や，Extensible Markup Language（XML）などで記載された定義・設定ファイル，ドキュメントなどである．さらに，ソースファイルから設定ファイルへの関連など，異なる種類の成果物間の関連を求めることもある．本研究では，プログラム要素間，特に，関数から関数への関連を取り扱う．

ロジカルカップリングの分析方法として，版管理システム（Version Control System，VCS）に格納された変更履歴に対してアソシエーション分析（バスケット解析）[16] を用いる方法を，**図 1** を用いて説明する．なお，一般に

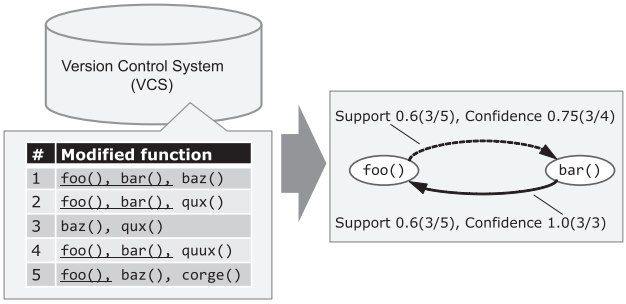

図 1 ロジカルカップリング分析の例
Fig. 1 Example of logical coupling analysis.

版管理システムに格納される変更履歴はソースファイル単位であるが，ここでは関数単位の識別ができているものとする．

アソシエーション分析においては，解析を行う単位をアイテム（item），同時に出現したアイテムのセットをトランザクション（transaction）と呼ぶ．ここでは，関数がアイテム，1コミットで変更された関数のセットがトランザクションとなる．たとえば図1に示す5コミットを入力とした場合，「関数 `bar()` を変更すると関数 `foo()` もまた変更される」という相関ルール（association rule）が得られる．この関係を「$\{$`bar()`$\} \Rightarrow \{$`foo()`$\}$」と表記する．本論文では，単一要素から単一要素への関係を考える場合には，集合表記を略して「`bar()` $\Rightarrow$ `foo()`」のように表記する．

相関ルールは次に示す3種類の指標で評価され，あらかじめ定められた閾値を超えたもののみが相関ルールとして検出される．X，Y をアイテム集合，D をトランザクションセット，$\sigma(X)$ を，X を含むトランザクション数としたとき，

支持度（support）：X と Y を含むトランザクションが全体の中に占める比率で定義され，相関ルールの出現頻度を示す．

$$\mathrm{supp}(X \Rightarrow Y) = \frac{\sigma(X \cup Y)}{|D|} \tag{1}$$

確信度（confidence）：X と Y を含むトランザクションの数を，X を含むトランザクションの数で割ったものとして定義され，相関ルールの確からしさ（再現率）を示す．

$$\mathrm{conf}(X \Rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)} = \frac{\mathrm{supp}(X \Rightarrow Y)}{\mathrm{supp}(X)} \tag{2}$$

リフト（lift）：確信度を Y の支持度で割った値で定義され，X の出現により，Y の出現確率がどの程度上がるかを示す．

$$\mathrm{lift}(X \Rightarrow Y) = \frac{\mathrm{conf}(X \Rightarrow Y)}{\mathrm{supp}(Y)} \tag{3}$$

たとえば，図1に示すトランザクションに対し，支持度0.5，確信度0.7という閾値を定めて検出したとすると，次の2種類の相関ルール（ロジカルカップリング）が得ら

れる．

- `foo()` ⇒ `bar()`：支持度 0.6；確信度 0.75；リフト 1.25
- `bar()` ⇒ `foo()`：支持度 0.6；確信度 1.00；リフト 1.25

3.3 予備調査

手法の議論に先立ち，製品ソフトウェア開発現場でのリポジトリの利用状況に関する調査を行った．対象リポジトリは 4 章で扱う組込み製品 A，B のリポジトリである．調査方法として版管理システムおよびチケット管理システム（Bug Tracking System，BTS）の目視調査および開発者に対するヒアリングを実施した．

VCS へのコミットは，ビルドが通る状態かつ，BTS の関連するチケット ID をコミットメッセージに入れることが規約で定められ，開発者の遵守状況も良い．一方，それ以上のことは製品やチーム，開発者依存である．たとえば，コミットの説明は総じて簡単であり，チケット ID 以外何も書かれていないコミットも散見される．また，1 つのコミットに複数のチケット ID が記載されていることもある．なお，規約はいわゆる trunk（master）ブランチに対してのみ存在し，開発ブランチの運用はチーム依存となっている．

ここで，チケットは不具合もしくは機能追加を管理するものであり，その粒度にはばらつきがある．多くのチケットでは数ファイル程度だが，大きな機能追加チケットでは数十ファイルが変更されることもある．また，製品 B では，不具合もしくは機能追加要件のチケットそれぞれについて，それを具体的な変更要求としてチームへ割り振るための子チケットが作成される．チームはおおむねモジュール[*1]と 1 対 1 対応となるため，複数モジュールにまたがる変更は，VCS 上はモジュールごとに分かれた変更となる．また，製品 B のチケット管理においては，ある不具合を引き起こした変更がチケットレベルで特定できているときには，起因元としてその ID を記入する欄がチケットに存在する．変更漏れの追跡に活用可能であるが，その記入は必須ではなく，開発者や不具合の重要度に依存する．

3.4 手法の概要

ロジカルカップリング分析を用いたアンチパターン検出手法の流れを**図 2** を用いて説明する．

(1) リポジトリ解析による変更履歴の取得：版管理システムへのプログラムの格納粒度はファイル単位であるのに対し，アンチパターン検出は関数単位で行いたい．そのため，履歴における変更差分と，各リビジョンのスナップショットに対するソースコード解析結果を付き合わせることにより，関数単位の変更履歴を得る．

(2) 変更履歴からのトランザクション生成：アソシエーション分析の入力となるトランザクションセットを生成する．このとき，マージ元ブランチの追跡および同一趣旨のコミットの併合を行う．詳細は 3.5 節で述べる．

(3) アソシエーション分析：生成したトランザクションに対して Apriori アルゴリズムを適用し，相関ルールを得る．詳細は 3.6 節で述べる．

(4) アンチパターン検出および可視化：相関ルールと，関数などプログラムエンティティの情報およびコードクローン解析結果を付き合わせることで，ロジカルカップリンググラフとして可視化する．このとき，グラフパターンマッチによりアンチパターンに合致する箇所をハイライトするとともに，同時変更に影響するプログラム構造上の関連を持つ箇所をハイライトすることで，結果の解釈を助ける．詳細は 3.8 節で述べる．

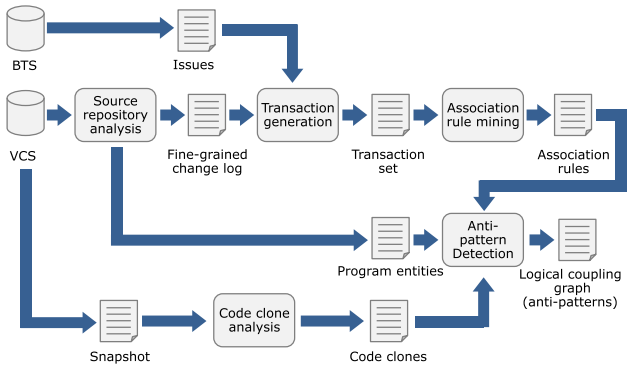

図 2 ロジカルカップリング分析を用いたアンチパターン検出
Fig. 2 Anti-pattern detection using logical coupling analysis.

3.5 変更履歴からのトランザクション生成

トランザクション生成において，基本的には，1 つのコミットで変更された関数群を 1 つのトランザクションとして扱う．ただし，予備調査で明らかになったように，1 つのコミットが 1 つの変更とならず，解析に悪影響を及ぼすことがある．そこで変更管理情報を用いることで同一要因に基づく変更を併合するとともに，マージの対処として，マージコミットの除去およびマージ元の追跡を行う．以下，それぞれについて説明する．

3.5.1 チケット管理のモデル

本論文で扱う，チケット管理に関連する概念的なモデルを**図 3** に示す．関数（Function）を変更するコミット（Commit）は何らかのタスク（Task）に基づいて行われる．タスクは課題（Issue）の配下として存在する．課題は機能追加（Feature）および不具合（Bug）に分類される．課題およびタスクは管理要素（Tracking Item）として抽象化され，変更を引き起こす元となった要素（たとえば，不具合から，その不具合を引き起こした機能追加）への参照を起因元（origin）として持つ．また，管理要素には一意な ID が付与されているものとする．

[*1] 本論文では，複数のソースファイルをグループ化する単位を意味する．

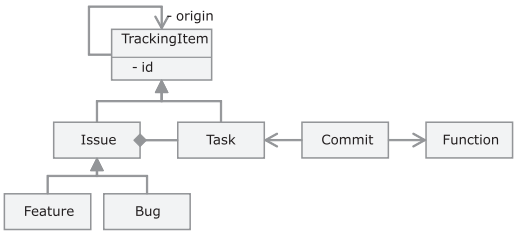

図 3 チケット管理のモデル
Fig. 3 ticket management model.

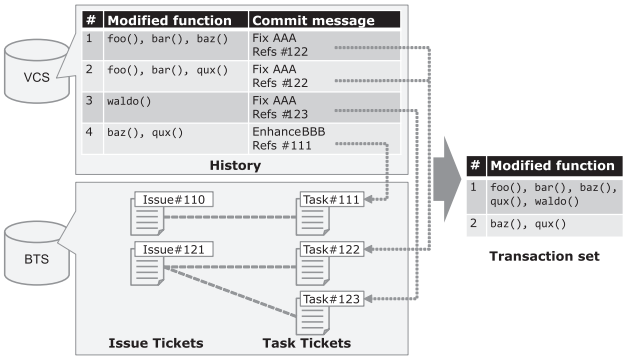

図 4 トランザクション生成
Fig. 4 Transaction generation.

3.5.2 トランザクション生成

本モデルに基づくトランザクション生成について図 4 を用いて説明する．トランザクションの生成に先立ち，コミットメッセージに記載された ID と付き合わせることで，コミットとチケットとの関連付けを行う．コミットメッセージ中には決められた書式にて ID が記入されているものとする．図 4 の例では，commit#1 中の “Refs #122” という記載より，task#122，およびその親となる issue#121 と，commit#1 が関連付けられる．この関連付け情報を用いることで，Issue チケット単位のトランザクションを生成する．図 4 の例では，Issue チケット#121 に関連付く commit#1，2，3 から transaction#1 が，Issue チケット#110 に関連付く commit#4 から transaction#2 が生成される．さらに，チケットに対して起因元が指定されていれば，その起因元のチケットのトランザクションへと併合する．

また，チケットへの対応付けができない場合も，24 時間以内に，同一ユーザから，同一メッセージにてコミットされた変更は同一要因と見なして併合する．

3.5.3 マージの処理

別ブランチの変更をマージするコミットは，複数の要因の変更を含むことが多くそのままトランザクションとして扱うとロジカルカップリング分析に悪影響を及ぼす．そのため，マージそのもののコミットを無視する代わりに，マージ元の変更をトランザクション生成に用いる．Git リポジトリにおいてはコミット列そのものがマージされるため特別な処理は不要だが，Subversion においてはマージされたブランチでの変更内容がまとめられたコミットが生成されるため，マージ元ブランチを特定し，スイッチする必要がある．

また，履歴上マージではないコミットであっても，手作業でのファイルコピーでのマージや，複数の変更要因を含む大規模コミットは，解析に悪影響を与える見込みが高いため除外する．具体的には，以下の条件となる．

- コミットによる変更ファイル数が閾値 α 以上
- 関連付くチケット ID の個数が閾値 β 以上

Moonen らによるアソシエーション分析に基づく変更予測に関する調査によると，小さいコミットの方が有益な情報を含み，α の閾値は 6 から 10 が良い性能となる [15]．β に関しては，複数のチケット ID は異なる趣旨の変更の混在を意味するため，できる限り小さい値が望ましいと考えられる．ただし，リポジトリ運用によってはコミット数を減らしすぎる可能性があるため，実際のコミットログの調査に基づき調整する必要がある．

3.6 アソシエーション分析

得られたトランザクションセットに対してアソシエーション分析を実施することで，相関ルールを得る．アソシエーション分析のアルゴリズムとしては Apriori アルゴリズム [16] を用いる．版管理システムに格納された変更履歴の特性を考慮し，Apriori アルゴリズムの閾値は次のような考え方で定める．

支持度（support）：支持度の定義は全体を分母とした割合（$\mathrm{supp}(X \Rightarrow Y)$）であるが，全体の規模に従い値が大きく異なってくるため，本論文では，出現数の絶対値 $\sigma(X \Rightarrow Y)$ を支持度の閾値として考える．ここで，関数やクラスといったプログラム要素の変更は，ごく一部の要素に極端に偏る一方で，ほとんどの要素は数回程度しか変更されないという性質を持つ．また，これは対象とするソフトウェアの部位や範囲によっても偏る．これらのことから，支持度の閾値として 3〜7 程度の低めの値を設定する．

確信度（confidence）：アンチパターン検出の意図としては，高い確率で同時に変更するものを検出する必要があるため，確信度の閾値は高めに指定する．ただし，必然性のある相関ルールであっても，過去の変更漏れなどを原因として確信度が低下する可能性があるため，1 よりやや低めの数値（0.8 程度）とする．

リフト（lift）：単体で頻繁に変更される要素は稀であり，リフトは総じて高い値になる．基本的に固定値 (1) を与える．

また，ロジカルカップリンググラフの辺として扱うことを考慮し，検出される相関ルールの左辺および右辺はそれぞれ 1 要素に限定する．

3.7 コードクローン解析

Duplicated Code 検出のため，リポジトリから得たソースコードスナップショットに対しコードクローン解析を行う．本論文ではトークンベースの Type-2 クローンを検出可能な CCFinderX [*2] を用いる．

提案手法では，クローンとして検出された個別のコード辺そのままではなく，関数間にどの程度クローンが存在するかという指標として利用するため，計算可能な範囲で，最小トークン数などのパラメータを小さめに設定する．

3.8 ロジカルカップリンググラフによるアンチパターン検出

アソシエーション分析により得られた相関ルールに基づき，ロジカルカップリンググラフを構築する．ロジカルカップリンググラフは，**図 5** に示すような，関数を頂点・相関ルールを有向辺とした有向グラフである．頂点および辺の属性として，検出されたアンチパターンの情報を持つ．また，理解を助けるための情報として，ファイルやモジュールでサブグラフを形成するとともに，頂点および辺の属性として参照関係やコードクローン関係など，プログラム構造上の関連を持つ．

本論文で検出対象としたアンチパターンについて，具体的な定義を以下に示す．

- Shotgun Surgery：ある変更が他の多数の小さな変更を引き起こす関数であり，ロジカルカップリンググラフにおいては，出力辺が多く，かつ，その先が多岐にわたる頂点として示される．具体的には，次の条件をすべて満たす頂点である．
  - 出次数が閾値 x 以上
  - 出力辺のうち少なくとも 1 つは異なるファイル（サブグラフ）に属する
- The Blob：本論文では様々な箇所の変更の影響を受けるアンチパターンとして定義しており，Shotgun Surgery とは逆に，多くの入力辺を持つ頂点としてあらわれる．具体的には，次の条件を満たす頂点である．
  - 入次数が閾値 y 以上
  - 出力辺のうち少なくとも 1 つは異なるファイル（サブグラフ）に属する
- Duplicated Code：本論文では，コードが重複し，かつ同時に変更される関数の対として定義している．具体的には，次の条件を満たす辺である．
  - 辺の両端の関数の対について，共通して存在するコードクローンのコード片により，関数が覆われる割合が閾値 z 以上

なお，図 5 では，次のようなハイライトにより頂点や辺の属性を示している．

[*2] http://www.ccfinder.net/（2019 年 10 月確認）

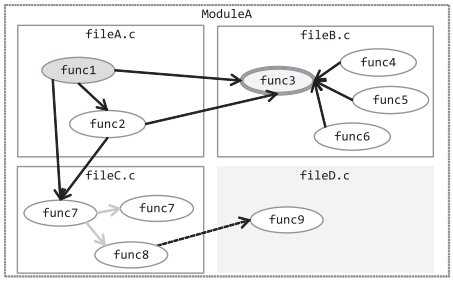

図 5 ロジカルカップリンググラフ
Fig. 5 Logical coupling graph.

- 変更の分散（Shotgun Surgery）：濃いグレーの頂点
- 肥満児（The Blob）：二重線枠の頂点
- サブグラフをまたがる辺・変更の分散として検出された頂点の出力辺・肥満児として検出された頂点の入力辺：濃色
- それ以外の辺：淡色
- 確信度が 1 の辺：実線
- 確信度が 1 未満の辺：破線
- 解析範囲のファイル：実線枠の矩形
- 解析範囲外のファイル：グレー塗りつぶしの矩形
- モジュール：点線の矩形

4. 適用実験

ソフトウェアのリファクタリングすべき箇所が検出できるかどうかを評価するため，オープンソースソフトウェア（OSS）および組込みシステム製品のソフトウェアそれぞれに対して提案手法を適用した．提案手法を実装したツールは VCS として Git および Subversion，言語として C, C++，BTS として Redmine，JIRA および 4.2 節で述べる内製 BTS システムにそれぞれ対応している．

なお，解析における閾値は，特記なければ下記の値を共通して用いるものとする．

- トランザクション生成：α：10；β：3
- アソシエーション分析：支持度：5；確信度：0.8
- アンチパターン検出：x：5；y：5；z：0.8
- コードクローン検出：最小トークン数 50，トークン種類 9

4.1 オープンソースソフトウェア

歴史が長い C 言語のソフトウェアとして Apache HTTP Server [*3] へ提案手法を適用する．解析設定について以下に示す．なおチケット情報を用いず解析を行った．

- 解析期間 2011-11–2016-11（7,012 コミット）
- 規模：487 ファイル，コメント抜き行数 197 K LOC（2016-11 時点）

得られたロジカルカップリンググラフの抜粋を**図 6** に示

[*3] https://httpd.apache.org/，Git リポジトリ https://git.apache.org/httpd.git

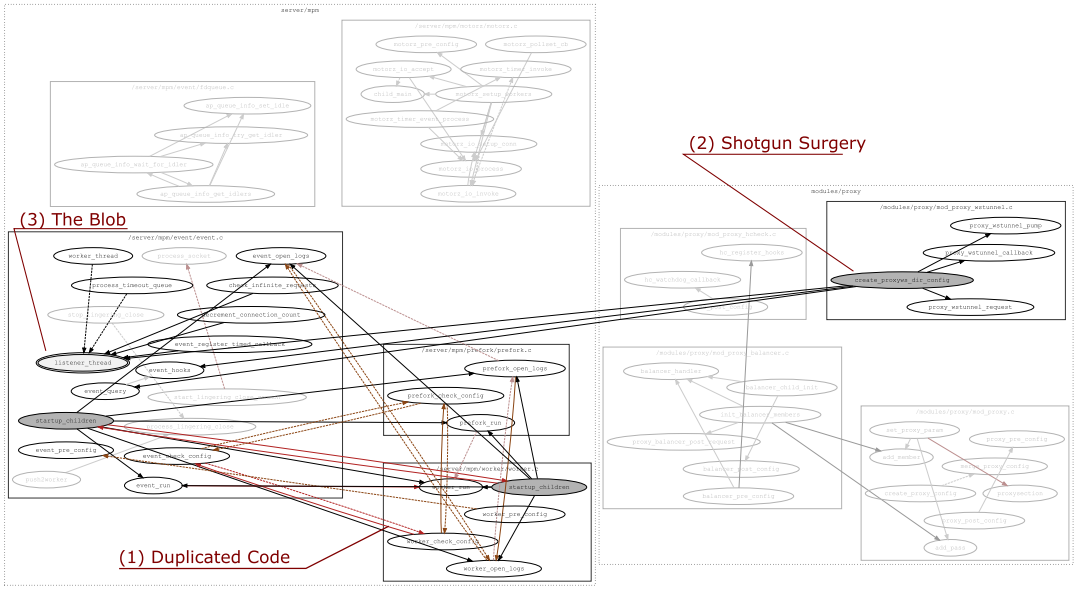

図 6 Apache HTTP Server のロジカルカップリンググラフ
Fig. 6 Logical coupling graph of Apache HTTP Server.

し，特徴的な部分についてソースコードを調査した結果を以下に述べる．

Duplicated Code (1)：異なるモジュール（`event`, `prefork`, `worker`）に属する同名/類似した名前の関数群が Duplicated Code として検出されている．これらのモジュールはサーバのプロセスの異なる管理方法を実装したものであり，コピーペーストにより実装された部分がそのまま一貫性を持って変更されていることを示している．

Shotgun Surgery (2)：関数 `create_proxyws_dir_config()` を中心とした Shotgun Surgery が検出された．この関数はモジュール `proxy` に属し，モジュール `event` に属する関数 `event_hooks()` や `event_query()`, `listener_thread()` とは直接の参照関係を持たないが，イベントの送受信を通じて間接的に結合している．

The Blob (3)：関数 `listener_thread()` が The Blob として検出されている．この関数はイベント処理を行う関数であり，イベントの処理で切り分ける巨大な switch 文を持つ．これより散在するイベント送信元との間接的な結合が生じている．

4.2 組込みシステム製品

以下の 2 種類の組込みシステム製品に対して適用した．

- 製品 A：C++言語約 500 K LOC；Subversion リポジトリ/内製 BTS システム
  - 解析期間 2011-02–2016–01（約 2,000 コミット）
  - 解析閾値：α：80；β：30
- 製品 B：C 言語約 300 K LOC；Subversion リポジトリ/内製 BTS システム
  - 解析期間：2014-03–2016-05（約 4,000 コミット）
  - 解析閾値：製品 A と同様

表 1 検出結果
Table 1 Detection result.

Target	# of LC	# of SS	# of Blob	# of DC
A	556	26	21	34
B	70	9 (3)	7 (1)	22 (22)

解析閾値に関し，コミットログの目視確認により，全体的に 1 コミットあたりの変更量が大きい傾向が見られたため，α および β の値を，特に大規模な一括コミットを弾くように設定した．なお，製品 B は適用対象を特定のモジュールに絞っている．ただし，モジュールをまたがった結果を得るため，製品全体を解析したうえで対象のモジュールへと結果を絞り込んだ．上記は絞り込み後の数値を示している．

検出されたロジカルカップリングの数（アソシエーションルールの数），Shotgun Surgery（SS），The Blob，Duplicated Code（DC）の数をそれぞれ**表 1** に示す．

製品 B については検出箇所それぞれに対して適合性を評価し，適合した個数を括弧内に示している．なお，ここでの適合性は，それぞれのアンチパターンの示す現象が実際に顕在化したものであるかどうかの観点から評価した．

以下，製品 A，B それぞれについて，特徴的な箇所についてソースコード調査および開発者ヒアリングを行った結果を述べる．

4.2.1 製品 A

Duplicated Code：外部インタフェースの実装を行っている部分のうち，インタフェース仕様のバージョン違いがそれぞれ別のクラスに実装されている箇所が Duplicated

Code として検出された．これは意図的な重複であり，その集約の是非の判断は難しいが，変更時に変更漏れのリスクが存在するため，コメントでの明示など追跡可能とする手段が必要である．

Shotgun Surgery：リソース管理にかかわる関数群が Shotgun Surgery として検出された．複雑な状態判定を行う，似て非なる条件分岐が散在している．ハードウェア変更などによりそれぞれの条件を個別に見直し・修正する必要があり，開発者には，影響分析および修正の難易度が高く，構造の見直しが必要な箇所として認識されている．

The Blob：通信にかかわる関数群のなかでデータ転送処理を行う関数が The Blob として検出されている．この関数構造もまた開発者に見直しが必要と認識されている．

4.2.2 製品 B

Shotgun Surgery/The Blob：ロジカルカップリングが密集し，Shotgun Surgery および The Blob が集中的に検出された箇所が存在した．これはデータテーブルの操作関数群であり，テーブル構造を変更する際にはこれら複数の関数を所定の手順に従い修正するため，検出された．開発者には，定型化した処理であり誤りにくいものの，変更の趣旨に対して修正工数が多いという問題をかかえる箇所として認識されている．一般に，変更時の影響確認のために関数の参照関係を追うことがあるが，Shotgun Surgery として検出された関数のうち 2 つの関数については，他関数の参照数（callee 数）が 0 および 1 であり，影響範囲の小さな関数のように見える．また，The Blob のある関数についても，被参照数（caller 数）は 2 であり，影響を受けやすい関数としては扱われにくい．

適合しなかったものは，すべて，たまたま修正が連続した関数の集合が存在し，かつ，そのうちのいくつかが Duplicated Code と関係する関数を含んだため入次数もしくは出次数が大きくなり検出されたものであった．影響の広さ/受けやすさの観点ではは誤りではないが，その主要因は Duplicated Code であるため，今回は非適合とした．

Duplicated Code：Apache HTTP Server でのそれと類似した形態のものが検出された．これは製品モデルごとの構成の違いを，ソースファイルを切り替えることで吸収している箇所であり，ほぼ同一の関数の組合せが同時に変更されている．変更の入り方が事前に見通しにくいことによる，意図的な重複である．ただし，ロジカルカップリングとして検出されたことは，この部位はモデルによって異なる変更が加わることが少ないことを示しており，共通化による効率化が見込まれる．検出された 22 辺すべてがこれに該当した．コードクローン解析結果のみを用いた場合ははるかに多くの関数対が候補となるが，その中から改善効果の見込まれるものを効率的に抽出できた．

5. 考察

5.1 リファクタリングに向けて

適用実験の結果として開発者の問題意識に合致したものが得られた．提案手法は，アンチパターンの中でも，特にプログラム要素間の関連に着目したものである．関数単体では妥当な規模や複雑度に抑えられていても，関連性に課題が存在し，複雑な変更が多数必要となることがある．これらの関連性に関する課題をプログラム解析のみで検出するには，データフローなどの観点から間接的な関連を多段に追跡する必要があり，技術的難易度やスケーラビリティが問題となる．さらに，解析自体が可能であっても，結果が大量になり，的確に絞り混むことが困難である．履歴を用いることで，課題のある関連を容易に検出し，開発者に気付きを与えることができる．

一方で，検出結果から改善につなげるためには，結果の解釈に関する課題を解決する必要がある．アンチパターンの中でも，たとえば「長すぎるメソッド」のようなものは，構造上の課題そのものを検出しているため，検出結果に対する改善指針が見えやすい．これに対し，ロジカルカップリング分析は履歴上の現象からの検出のため，その背景に存在する構造上の課題を別途明らかにしなければならない．文献 [4] では，共通する対処方法として「メソッドの移動」があげられているが，適用実験で検出された箇所は，いずれもアーキテクチャレベルの課題が存在すると見られ，局所的なリファクタリングでは解決困難と考えられる．提案手法では，コードクローン検出との組合せを実施したが，同様に他の要因についても分析実施し，より細分化された状況へと絞り込むことが求められる．

さらに，実際にリファクタリングするかどうかに関しては，提案手法自体はバッドスメルから定性的に悪影響を示すのみであり，実施判断は開発者の経験や勘，勇気に依存する．より戦略的にリファクタリングを進めるためには，悪影響を，より具体的・定量的に示す必要があると考えられる．たとえば，アンチパターンの検出箇所と，過去の不具合のつながりが示されれば，品質にかかわる問題として扱うことができる．もちろん，不具合の誘発は技術的負債の一面であるため，それのみに頼った動機付けは偏った施策となるため適切とはいいがたいが，リファクタリング含め対策指針を明確にするためには，何らかの方法での定量化が必要となると考えられる．

5.2 開発環境の変遷への対応

開発者へのヒアリングのなかで，ハードウェアプラットフォームの変更のたびに同じ変更の繰り返しがあるため，それを検出できないかという意見があがっていた．しかし，結果としてそのようなロジカルカップリングは検出されなかった．その要因の 1 つとして，製品開発の歴史のな

かで，開発環境，特にリポジトリそのものも発展していることが考えられる．たとえば，製品 B では，10 年以上，開発者による手作業での版管理が行われてきたが，何年か前に Subversion を経て Git（GitLab）へと移行した．本適用実験の時点で，年単位の履歴が蓄積されており，これは文献 [15] で示された目安を超える十分な量である．しかし，製品世代としては 1 つの世代に閉じているため，ハードウェア変更対応のような，世代をまたがって繰り返されるような変更は検出されなかったと考えられる．

過去のスナップショットを履歴として扱うことで提案手法の適用そのものは可能であるが，変更の粒度の違いに加え，ソフトウェアの構成の変化により変更履歴に現れる関数自体が異なるため，妥当なアンチパターンが検出されるかどうか未知数である．そのため，世代をまたがった解析は，世代に閉じた解析と目的を分けてアプローチ検討する必要があると考えられる．

5.3 妥当性への脅威

適用実験では特定の限られたソフトウェアでのみ実験を行っている．また，解析期間の設定や，ロジカルカップリング分析にかかるパラメータに関しても単一の結果のみを示している．適用実験にて提案手法，すなわちアンチパターン検出に対するリポジトリマイニングの可能性を実証することができたが，この結果を一般化するためには，より幅広いソフトウェアでの適用実験や，実験設定を変えた場合の知識の蓄積が必要である．

また，検出された箇所について，それが何らかの問題を持つかどうか，すなわち適合性をソースコード調査およびヒアリングによって検証を行った．ソースコード調査に基づく判断は著者らが行ったが，何らかの客観的な基準に基づいたものではなく経験などの影響を受けた主観的なものである可能性がある．製品ソースコードについては開発者ヒアリングも実施したが，開発者にとって実際どの程度の認識となっているかは不明である．また，検出の再現性，すなわち検出すべきものがすべて検出できているかどうかでの観点での評価は行えておらず，今後の課題となっている．

6. まとめと今後の課題

本論文では，リポジトリマイニングにより，ソフトウェアのリファクタリングすべき箇所をアンチパターンとして自動的に検出する手法を提案した．提案手法は，組込みソフトウェア向けに，変更の分散（*Shotgun Surgery*），肥満児（*The Blob*），重複したコード（*Duplicated Code*）の 3 種類のアンチパターンを，ロジカルカップリング分析とプログラム解析の組合せにより検出する．適用実験として，歴史が長い OSS ソフトウェアである Apache HTTP Server および 2 種類の組込みシステム製品のソフトウェアからアンチパターンを検出した．組込みシステム製品に関しては開発者ヒアリングを実施し，開発者の問題意識に合った検出結果であることを確認した．

本研究は，組込みソフトウェア開発において効果的なリファクタリングを進めるための取り組みの一環であり，今後の課題として，アンチパターンのより具体的な状況の分析および提示，適用すべきリファクタリングの推薦があげられる．また，対象を拡大した定量的な評価があげられる．

参考文献

[1] Mens, T. and Tourwé, T.: A Survey of Software Refactoring, *IEEE Trans. Softw. Eng.*, Vol.30, No.2, pp.126–139 (2004).

[2] Chidamber, S.R. and Kemerer, C.F.: A Metrics Suite for Object Oriented Design, *IEEE Trans. Softw. Eng.*, Vol.20, No.6, pp.476–493 (1994).

[3] Ó Cinnéide, M., Yamashita, A. and Counsell, S.: Measuring Refactoring Benefits: A Survey of the Evidence, *Proc. 1st International Workshop on Software Refactoring, IWoR 2016*, pp.9–12, ACM (2016).

[4] マーチン・ファウラー：リファクタリング—プログラミングの体質改善テクニック，ピアソン・エデュケーション (2000).

[5] ジョシュア・ケリーエブスキー：パターン指向リファクタリング入門—ソフトウェア設計を改善する 27 の作法，日経 BP 社 (2005).

[6] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).

[7] 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, pp.3_29–3_42 (2011).

[8] Kim, M., Zimmermann, T. and Nagappan, N.: A Field Study of Refactoring Challenges and Benefits, *Proc. ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, pp.50:1–50:11, ACM (2012).

[9] 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, pp.1_14–1_29 (2014).

[10] Moha, N., Gueheneuc, Y.G., Duchien, L. and Meur, A. F.L.: DECOR: A Method for the Specification and Detection of Code and Design Smells, *IEEE Trans. Softw. Eng.*, Vol.36, No.1, pp.20–36 (2010).

[11] Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Poshyvanyk, D. and Lucia, A.D.: Mining Version Histories for Detecting Code Smells, *IEEE Trans. Softw.*

Eng., Vol.41, No.5, pp.462–489 (2015).
[12] Zimmermann, T., Zeller, A., Weissgerber, P. and Diehl, S.: Mining Version Histories to Guide Software Changes, *IEEE Trans. Softw. Eng.*, Vol.31, No.6, pp.429–445 (2005).
[13] 堀 旭宏，市井 誠，川上真澄：リポジトリマイニングに基づく「長すぎるメソッド」検出手法，ソフトウェア工学の基礎ワークショップ（FOSE 2018）(2018).
[14] Ying, A.T.T., Murphy, G.C., Ng, R. and Chu-Carroll, M.C.: Predicting Source Code Changes by Mining Change History, *IEEE Trans. Softw. Eng.*, Vol.30, No.9, pp.574–586 (2004).
[15] Moonen, L., Alesio, S.D., Binkley, D. and Rolfsnes, T.: Practical guidelines for change recommendation using association rule mining., *Proc. ASE 2016* (2016).
[16] Agrawal, R. and Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases, *Proc. 20th Int'l Conf. Very Large Data Bases*, *VLDB '94*, pp.487–499 (1994).

市井 誠 （正会員）

2009 年大阪大学大学院情報科学研究科博士後期課程修了．同年株式会社日立製作所入社．現在，同社研究開発グループシステムイノベーションセンタ所属．博士（情報科学）．プログラム解析およびリポジトリマイニング，リファクタリングに関する研究に従事．IEEE Computer Society 会員．

川上 真澄 （学生会員）

1998 豊橋技術科学大学大学院工学研究科修士課程修了．同年株式会社日立製作所入社．現在，研究開発グループ主任研究員．現在，ソフトウェア工学研究と製品開発適用に従事．IEEE Computer Society 会員．