# 大規模ソフトウェアの保守支援を目的とした類似性分析と費用対効果見積りの研究

提出先　大阪大学大学院情報科学研究科

提出年月　2025 年 1 月

横井　一輝

# 論文一覧

## 主要論文

1. 横井 一輝，崔 恩瀞，吉田 則裕，井上 克郎，“情報検索技術に基づく細粒度ブロッククローン検出”，コンピュータソフトウェア，Vol. 35，No. 4，pp. 16-36，2018 年 10 月．（学術論文）
2. 横井 一輝，崔 恩瀞，吉田 則裕，松下 誠，井上 克郎，“情報検索技術と深層学習を用いたコード片類似性判定法の比較調査”，電子情報通信学会論文誌 D，Vol. J106-D，No. 4，pp. 231-243，2023 年 4 月．（学術論文）
3. Kazuki Yokoi, Eunjong Choi, Norihiro Yoshida, Joji Okada, Yoshiki Higo, “Cost-Benefit Analysis for Modernizing a Large-Scale Industrial System”, Proceedings of the 30th Asia-Pacific Software Engineering Conference (APSEC 2023), pp. 441-449, December 2023.（国際会議録）

## 関連論文

1. 太田 悠希，吉田 則裕，崔 恩瀞，槇原 絵里奈，横井 一輝，“マイクロサービスにおけるコードクローンの言語間分析”，日本ソフトウェア科学会 第 31 回ソフトウェア工学の基礎ワークショップ，2024 年 11 月．（国内会議録）

# 内容梗概

社会におけるソフトウェアの重要性が高まってきた現在では，特に社会基盤や企業の基幹業務を支える大規模かつ複雑なソフトウェアを保守し，品質を保つことが重要である．ソフトウェア保守は，ソフトウェアの全ライフサイクルにおいて多くのコストを占めるため，その支援が必要不可欠である．また，長年にわたり保守を繰り返して老朽化したレガシーシステムでは，ソフトウェアの変更や修正に多大な時間とコストがかかり，ビジネスニーズに対応しきれなくなる．そのため，ソフトウェア保守よりも広範囲な変更を伴うモダナイゼーションを実施し，ソフトウェアシステムを進化させる必要がある．

ソフトウェア保守やモダナイゼーションを効率的に進めるには，開発者がソフトウェアの性質や振る舞いを十分に理解することが求められる．しかし，ソフトウェアの規模や複雑さが増大するにつれて，手作業による十分な理解が難しくなる．そのため，近年のコンピュータの計算能力向上を背景に，ソフトウェア保守を支援するためのソースコード静的解析が盛んに研究されている．

本論文では，ソフトウェア保守の効率を低下させる要因の 1 つであるコードクローンの把握を支援する目的で 2 つの研究を，モダナイゼーションの推進を妨げる要因の 1 つである費用対効果の試算を支援する目的で 1 つの研究を実施した．

1. コードクローンの把握を支援
   (a) 情報検索技術に基づく細粒度ブロッククローン検出
   (b) 情報検索技術と深層学習を用いたコード片類似性判定法の比較調査
2. モダナイゼーションの費用対効果の試算を支援
   (a) 段階的再構築における依存関係分析を用いた費用対効果の試算

1-(a) については，構文解析によりコードブロックを抽出し，情報検索技術を利用して，意味的に処理が類似したブロッククローンの検出手法を提案した．これにより，既存のコードクローン検出手法よりも高精度でコードクローンを検出でき，さらに保守作業を行いやすいコードクローンを検出することが可能となった．また，クラスタリング手法や特徴ベクトルのデータ構造を工夫することで，既存手法よりも高速かつ低メモリ消費で検出できた．

1-(b) については，コード片の処理内容の意味的な類似性を高精度かつ高速に判定す

るため，情報検索技術と深層学習の効果的な組み合わせを調査した．調査の結果，情報検索技術の一種である LSI（Latent Semantic Indexing）と深層学習モデルの組み合わせが高精度で判定可能であり，さらに実行速度も最も速いことが確認された．

2-(a) については，過去に段階的再構築を実施した大規模な金融システムを対象に，依存関係分析を用いた費用対効果試算手法を適用し，その妥当性を検証した．モダナイゼーションの失敗リスクを軽減するためには，システムの一部を切り出す段階的再構築が有効とされる．しかし，段階的再構築の費用対効果を試算することは難しく，著者が所属する企業においても計画当初の見積りと実績値に乖離が生じた事例がある．そこで本研究では，システムの規模と依存関係の情報を活用し，費用として段階的再構築に必要な工数を，効果として削減可能な保守開発工数を試算し，実績値との乖離を比較した．また，プロジェクトの関係者にヒアリングを行い，適用手法の妥当性を定性評価した．評価の結果，費用試算は実績値との乖離が小さい一方で，効果試算は実績値との乖離が大きかった．関係者ヒアリングの結果では，費用試算の結果に対してはやや妥当との意見が得られたが，効果試算の結果に対しては妥当性に疑問を持つ意見が多かった．また，試算モデルに対する納得感は中立的な意見が多く，今後の適用可能性については肯定派と否定派に分かれた．

これらの研究により，大規模ソフトウェアの保守およびモダナイゼーションという社会基盤を維持するうえで欠かせない活動に対し，一定の効率化を果たすことができたと考える．

# 謝辞

本研究を行うにあたり，日頃から様々なご指導を賜りました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 肥後芳樹教授に，心から感謝申し上げます．

本論文を執筆するにあたり，様々なご指導ご助言を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 楠本真二教授，ならびに増澤利光教授に感謝申し上げます．

本研究を行うにあたり，研究方針など様々なご指導ご助言を頂きました，立命館大学情報理工学部 吉田則裕教授，ならびに京都工芸繊維大学情報工学・人間科学系 崔恩瀞准教授に，心から感謝申し上げます．

本研究を行うにあたり，様々なご指導を頂きました，南山大学理工学部 井上克郎教授，ならびに大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠准教授に感謝申し上げます．

本研究を行うにあたり，研究活動の中でご助言を頂きました，ノートルダム清心女子大学情報デザイン学部情報デザイン学科 神田哲也准教授，奈良先端科学技術大学院大学先端科学技術研究科情報科学領域 嶋利一真助教に感謝申し上げます．

また，本研究の実施にあたり，株式会社 NTT データグループの皆様には研究の機会や実践の場を与えていただきましたことを感謝いたします．その中でも直接研究を支援し，様々なご助言を頂いた，株式会社 NTT データグループ 岡田譲二氏にこの場で御礼を述べさせていただきます．研究についてのご支援やご助言を頂きました株式会社 NTT データグループ 竹之内啓太氏，ならびに上田永樹氏にはこの場で御礼申し上げます．

最後に，日々の研究生活の中でご助言，ご協力を頂いた，大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座の皆様に厚く御礼申し上げます．

# 目次

# 図目次

# 表目次

# 第 1 章

# はじめに

## 1.1 ソフトウェア保守とその課題

現代社会において，ソフトウェアは様々な場所や用途で利用され，社会基盤や大企業の基幹業務を支える役割を担うなど，その社会的役割は大きくなっている．特に，高い信頼性と品質が求められる一方で，開発にかけられる時間や人的，計算機資源が限られているため，効率的かつ高品質なソフトウェアの開発および保守の重要性が増している．また，社会の変化やニーズに迅速に対応するためには，高い生産性でソフトウェアの開発および保守ができることも求められている．こうした背景から，ソフトウェア工学の分野でも，ソフトウェア開発と保守の品質向上や生産性向上のための支援が重要となっている．

ソフトウェア工学における重要な課題の 1 つとして，保守作業の効率化が挙げられる．ソフトウェアの保守作業とは，“ソフトウェアの納入後，ソフトウェアに対して加えられる，欠陥の修正，性能などの改善，変更された環境に適合させるための修正” のことを指す [1]．また，ソフトウェア保守は，その目的により以下の 4 つに分類されている [2, 3]．修正の分類とソフトウェア保守の分類の関係は図 1.1 に示す．

**是正保守** (corrective maintenance)　ソフトウェア製品の引渡し後に発見された問題を訂正するために行う受身の修正．この修正によって，要求事項を満たすようにソフトウェア製品を修復する．なお，是正保守の一部として，是正保守実施までシステム運用を確保するための，計画外で一時的な修正として，「緊急保守 (emergency maintenance)」がある．

**予防保守** (preventive maintenance)　引渡し後のソフトウェア製品の潜在的な障害が運用障害となる前に発見し，是正するための修正

**適応保守** (adaptive maintenance)　引渡し後変化した，または変化している環境において，ソフトウェア製品を使用できるように保ち続けるために実施するソフトウェア製品の修正．適応保守は，必須運用ソフトウェア製品の運用環境変化へ順応するために必要な改良を提供する．これらの変更は，環境の変化に歩調を

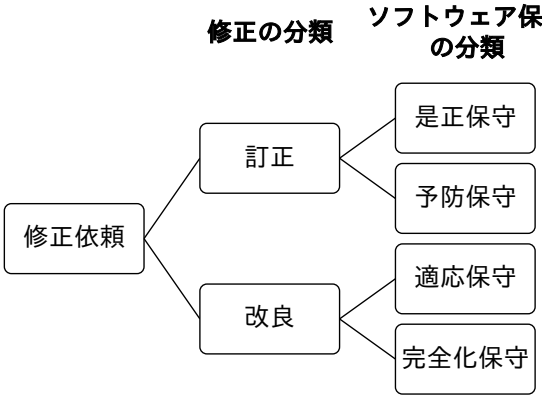


図 1.1　ソフトウェア保守の分類

合わせて実施する必要がある．例えば，オペレーティングシステムの更新が必要になったとき，新オペレーティングシステムに適応するためには，幾つかの変更が必要な可能性がある．これは，アプリケーションの全体機能要件は変わらないにも関わらず，オペレーティングシステムやミドルウェアの変更，ハードウェアの変更，法改正などに伴ってアプリケーションソフトウェアに影響する部分の改良が必要になるようなケースである．

**完全化保守** (perfective maintenance)　引渡し後のソフトウェア製品の性能または保守性を改善するための修正．完全化保守は，利用者のための改良，プログラム文書の改善を提供し，ソフトウェアの性能強化，保守性などのソフトウェア属性の改善に向けての記録を提供する．

ソフトウェア保守は，ソフトウェア開発に比べて長期間実施されることが多い．そのため，ソフトウェア保守にかかるコストはソフトウェアの全ライフサイクルにかかるコストの 3 分の 2 を占めており，保守作業にかかるコストは大きいことが知られている [4, 5, 6]．また，日本情報システム・ユーザー協会（JUAS）によると，21 年以上前に構築された基幹系システムの存在が報告されている [7]．これらのことから，まずはソフトウェアの保守をより効率的に行うことが重要である．

ソフトウェア保守を困難にする要因のひとつとして考えられているのが，コードクローンの存在である [8, 9]．コードクローンとは，互いに一致または類似したコード片のことである [10]．とりわけ実務におけるソフトウェア開発では，コードクローンをリファクタリングなどで除去するだけでなく，コードクローンの適切な管理が重要視されている [11, 12, 13]．例えば，あるコード片に欠陥が存在することを確認した場合，そのコード片のコードクローンにも欠陥が存在する可能性が高い．そのため，ソフトウェア保守の担当者は，欠陥が存在するコード片を修正した後，そのコード片のコードクローンを全て確認し，同様の修正をする必要がある．しかし，大規模ソフトウェアを保守する場合，全てのコードクローンを手作業で見つけることは困難である．

そのため，コードクローンの存在がソフトウェア保守のコスト増大につながっている．

また，コードクローンの中でも，処理内容が類似しているが構文的に類似していないコードクローンを検出し，適切に管理することの重要性も指摘されている [14]．実務においても，長年にわたって修正が繰り返された結果，類似しているが構文的に類似していないコードクローンが増えている．しかし，構文的に類似していないコードクローンは全く異なる書き方がされていることが多いため，既存のコードクローン検出法の多くは検出が難しい [15, 16]．したがって，構文的に類似していないコードクローンを検出することは，ソフトウェア保守の効率化において重要である．

## 1.2 モダナイゼーションとその課題

レガシーシステムとは，ビジネス上は重要だが，保守の継続が困難な巨大なメインフレーム上の基幹システムのことである [17]．レガシーシステムは企業の中核をなすシステムのため，その企業に現在でも多くの収益をもたらしており，その障害は日常業務に深刻な影響を与える．また，長期間正常に動作し続けてきた信頼性の高いシステムでもある．その一方で，レガシーシステムの多くは，20 から 30 年以上前に作られた古いシステムであり，時代遅れの技術を用いて長年保守し続けられたため，その変更には多くの期間とコストが必要になるという問題がある [18]．レガシーシステムは，ビジネスのコアなシステムが故に，新たなビジネス要求によって頻繁に変更され続けており，その結果，保守が困難な構造化されていないソースコードになっている．また，長年保守されているため，もともとのプログラマーが退職していたり，文書が古いままになっていたりする．これらの理由から，変更時にソースコードの調査や意図しない場所への影響調査が必要となり，その変更や修正には多くの期間とコストが必要になる．現在の急速に変化するビジネス環境では，組織内外の変化，法律や規制の変化など，様々な変化に迅速に対応しなければならない．レガシーシステムも同様にそういったビジネス環境の変化に追随して変更を迅速に行われる必要がある．こういった状況において，レガシーシステムの変更や修正に多くの期間とコストが必要になるのは，企業において致命的な問題である．このような問題があるにもかかわらず，1800 億行以上のレガシーコードが依然として使用されていると推定されている [19, 20]．

レガシーシステムの変更や修正に多くの期間とコストがかかる問題を解決するために，レガシーシステムのモダナイゼーションがしばしば行われる．レガシーシステムのモダナイゼーションとは，保守コストの削減と柔軟性の向上を目的に，ソフトウェアのコンポーネントやプラットフォームを交換，再開発，移行することで，既存のソフトウェアシステムを進化させるプロセスである．ソフトウェア保守を繰り返し，システムが古くなり最終的にソフトウェア保守がビジネスニーズに追い付かなくなった場合，ソフトウェア保守よりも広範囲な変更を伴うモダナイゼーションが必要になる．レガシーシステムがいまだに多くの企業で利用されているため，レガシーシステムのモダ

ナイゼーションは多くの企業の IT 戦略上重要な課題であり，レガシーシステムのモダナイゼーションサービスを提供する IT 企業も多い．調査会社の MarketsandMarkets 社によると，レガシーシステムのモダナイゼーションサービスの市場は 2024 年に 198 億米ドルとなり，2029 年には 396 億米ドルまで成長すると予測している [21].

しかし，産業界においてレガシーシステムのモダナイゼーションは十分に進んでいない．モダナイゼーションの推進を困難にする要因として，以下の点が指摘されている [18].

**時間的制約**　有識者やドキュメント不足などの影響を受け，計画通りの予算および期間で完了することが困難.

**費用対効果の見積り**　経営陣は，一度投資した際はその回収を優先し，モダナイゼーションの費用対効果の説明を求める．しかし，システム開発の初期段階で，モダナイゼーションの費用および効果を見積もるのは難しい.

**データ移行**　レガシーシステムの多くはリレーショナルデータベースを使用しておらず，最新のシステムへのデータ移行に失敗する可能性が高い.

**システムの複雑化**　その場限りの保守開発が繰り返された結果，レガシーシステムが複雑化しており，モダナイゼーションを難しくしている.

**知識不足**　技術知識として，レガシーシステム特有のデータベース，OS，ミドルウェアなどの専門的な知識が求められる．また業務知識として，レガシーシステムが担っている業務を理解している必要がある．しかし，ドキュメント不足がこれらの問題を一層深刻にしている.

**テストの困難さ**　システムの複雑化および知識不足により，テストケースの抽出が困難になり，モダナイゼーション前後のシステム機能を比較するテストに多くの時間がかかる.

**業務ロジックの特定と優先順位付け**　業務の有識者不足により，レガシーシステムの中からモダナイゼーション対象の業務ロジックを特定し，優先順位を付けることが困難.

**組織からの抵抗**　レガシーシステムの専門知識を持った技術者が不要になることを恐れるシステム開発者から協力を得にくい．また，変化を好まないシステム利用者も，新しい技術やユーザーインターフェースへの適応を嫌がる.

このように，モダナイゼーションの推進には技術面のみならずビジネス面にも課題があり，不具合が顕在化していないシステムを最新化する動機は生まれにくい．特に，時間的制約や費用対効果の見積りの難しさは，モダナイゼーションが進まない大きな課題である [18]．そのため，レガシーシステムには保守コストの増加，開発生産性の悪化，有識者不足といった問題があるにもかかわらず，モダナイゼーションは進んでいない現状がある.

## 1.3 ソフトウェア保守とモダナイゼーションの関係

モダナイゼーションはソフトウェア保守の一部として位置づけられ，「モダナイゼーション以外のソフトウェア保守」（以下，単に「ソフトウェア保守」と呼ぶ）と「モダナイゼーション」は密接に関連している．ソフトウェア保守は，ソフトウェア納入後の運用フェーズにおける反復的なソフトウェアの変更を指し，主に欠陥の修正や小規模な機能強化などが中心となる．本節では，ソフトウェア保守とモダナイゼーションの関係について述べる．

最初に 1.1 節で述べたとおり，ソフトウェア保守とは “ソフトウェアの納入後，ソフトウェアに対して加えられる，欠陥の修正，性能などの改善，変更された環境に適合させるための修正” を指す [1]．これは，ソフトウェアに対して小さな変更を繰り返し加えていく増分的かつ反復的なプロセスである．多くの場合，保守作業の大半は欠陥の修正や大きな構造変更を伴わない小さな機能強化で占められている [22]．また，独立行政法人情報処理推進機構では，「改修・保守」という用語を用いて，ソフトウェア保守を “納入後のシステムの運用フェーズで，ベースとなるシステムが存在し，機能追加など改修を伴う開発” と定義している [23]．このように，ソフトウェア保守は欠陥の修正や機能強化を目的とした，運用フェーズで継続的に実施される小規模な変更を含む作業と位置付けられる．

次に 1.2 節で述べたとおり，モダナイゼーションとは “保守コストの削減と柔軟性の向上を目的に，ソフトウェアのコンポーネントやプラットフォームを交換，再開発，移行することで，既存のソフトウェアシステムを進化させるプロセス” を指す [18]．ソフトウェア保守を継続することで，ある程度の期間はビジネスニーズに対応できるが，システムの老朽化が進むにつれて保守だけではビジネスニーズの増加に追随できなくなる．その際には，ソフトウェア保守よりも広範囲な変更を伴い，多くの時間と労力を要するモダナイゼーションが必要になる [22]．また，独立行政法人情報処理推進機構の定義では，このモダナイゼーションに相当する作業を「再開発」と呼び，“既存システムが存在し，機能仕様を殆ど変更する事無く，作り直す開発” としている [23]．すなわち，モダナイゼーションは保守コストの削減や柔軟性の向上を目的とした，ソフトウェアおよびシステム全般を大規模に更新する作業として位置付けられる．

図 1.2 にソフトウェア保守からモダナイゼーションへのライフサイクルを示す．この図では，縦軸をソフトウェアの機能量，横軸が時間の経過とし，破線は時間の経過とともに機能要求が増えていくことを，実線は実装されたソフトウェアの機能量を表している．図 1.2 の左側に示される開発フェーズでは，機能要求が満たされるまでソフトウェアが開発され，納入後は保守フェーズへと移行する．保守フェーズでは，増え続ける機能要求に対応するため，欠陥修正や機能追加などのソフトウェア保守が繰り返し行われる．しかし，時間の経過とともに保守性が悪化すると，ソフトウェアの

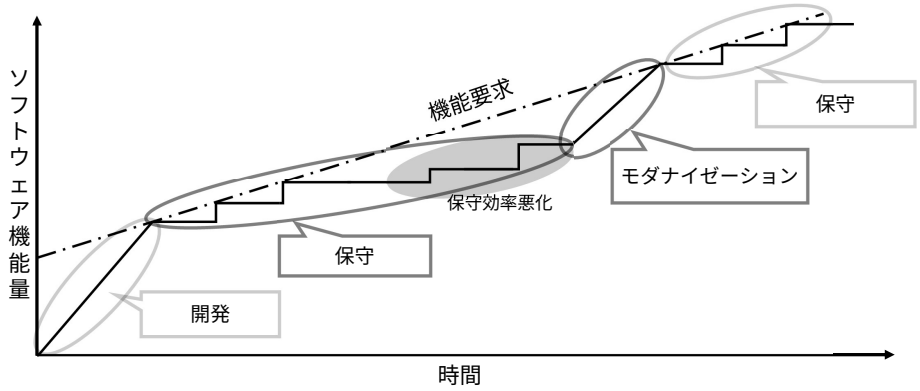


図 1.2　ソフトウェア保守からモダナイゼーションへのライフサイクル

**（注）** 文献 [22] の図 1-3 に基づき作成

保守のみでは増大する機能要求に追随できなくなる．そこで，より広範囲な変更を伴い，多くの時間と労力を要するモダナイゼーションが行われる．このように，ソフトウェア保守からモダナイゼーションへのフェーズの移行は連続的なものであり，厳密な境界の設定は困難である．しかし，変更の規模や目的に着目することで，両者を概念的に区別できる．

## 1.4　ソースコード静的解析

### 1.4.1　コンピュータを用いたソースコードの分析

1.1 節でも述べたとおり，長期間運用されるソフトウェアの開発において，保守作業にかかるコストが大きい．そのため，保守作業を支援する手法やツールの研究・開発が盛んに行われている．保守作業を支援する手法やツールの中には，コンピュータを用いてソースコードを分析し，保守作業に有用な情報を自動的に抽出するものが多い．開発者は，ソースコードのどこをどのように変更するか，また変更後にどのようなテストを行うべきかを決定するため，ソフトウェアを実際に動作させたり，ソースコードを読み込んだりすることで，ソフトウェアの振る舞いや呼び出し関係を分析する必要がある．しかし，ソフトウェアの規模が大きくなるにつれて，手動での分析は困難になる．そのため，コンピュータを用いたソースコードの分析手法が求められている．これらの手法やツールのうち，代表的なものを以下に示す．

**リバースエンジニアリング**　リバースエンジニアリングとは，システムの構成要素 (component) および構成要素間の関係を特定し，そのシステムを別の形式，もしくはより高い抽象度で表現することである [24]．ソフトウェア開発環境の中には，Imagix 4D[25] 等のようにソースコードからフローチャートやコール

グラフ（関数間の呼び出し関係を表すグラフ）を生成する機能を持つものや，Rational Software Modeler[26] 等のようにソースコードからクラス階層情報を抽出し，可視化する機能を持つものが存在する．また，リバースエンジニアリングを行う手法の一種として，設計の復元 (Design Recovery)[27] を行う手法が研究されている．設計の復元とは，設計に関する抽象概念 (Design Abstraction) をソースコードおよび他の情報（設計書や開発者の経験，対象とする問題とドメインに関する一般的な知識）から再現することである [27]．設計を復元する代表的なツールとして，ソースコード中からデザインパターン [28] の実装部分を自動的に特定するツール [29, 30] をいくつか挙げることができる．これらは，ツールの開発者もしくは使用者がデザインパターンに関する一般的な知識（デザインパターンが実装されている部分の構文的特徴など）をあらかじめ与えておくと，その知識に基づいてデザインパターンの実装部分を対象ソースコード中から特定する．このように，デザインパターンの実装部分を特定することは，保守作業を行ううえで有益であるとされている．例えば，保守対象のソースコード内で実装されているデザインパターンを明示すると，保守作業にかかる時間と混入する欠陥の数が減少したという実験結果が報告されている [31]．

**回帰テスト**　保守作業を困難にする要因の 1 つとして，ソースコードの一部を変更すると，変更部分だけでなく他の部分の振る舞いも変化する可能性が指摘されている [32]．

そのため，回帰テスト（変更後の振る舞いが要求を満たしているかを確認するためのテスト）では，変更部分のテストだけでなく，他の部分についてもテストを検討する必要が生じる．このことから，必要十分なテストの組み合わせを算出するための手法が数多く提案されている [32, 33]．

保守対象がオブジェクト指向プログラムの場合，Dynamic Dispatch（同じ型の参照型変数であっても，実行時におけるインスタンスの型に依存して呼び出される手続きが変化すること）が原因で，開発者にとって波及効果を理解することが難しくなる [32]．よって，一般的な手続き型プログラムと比較して，オブジェクト指向プログラムの方が，回帰テストにおいて実行すべきテストを適切に特定することが難しいと言える．この問題を解決するために，Chianti というツールが開発されている [32]．Chianti に，Java 言語で記述された変更前と変更後のソースコードおよび変更前のソースコード用に作られたテストコードの集合を与えると，入力したテストコードの中で，再度動作させるべきもののみを提示する．Chainti は，まず，変更前と変更後のソースコードについて，仮想メソッド（子クラスのメソッドがオーバライドできるメソッド）をオーバライドするメソッドの集合のそれぞれ算出する．そして，それらの差分を求めることで Dynamic Dispatch の変化を特定し，Dynamic Dispatch が変化する可能性のあるテストコードを提示する．

**ソースコード解析**　ソースコード解析とは，ソースコードの中身や動作を解析する技術の総称である．ソースコード解析の例として，メトリクス計測がある．ソースコードの保守性（保守しやすさ）を評価するメトリクスの代表的なものとして，CK メトリクス [34] が挙げられる．CK メトリクスは，オブジェクト指向プログラムに含まれるクラスを対象とした 5 つの複雑度メトリクスから構成されている．CK メトリクスとして，複雑度メトリクスが満たすべき数学的性質 [35] をおおむね満足していること [34]，加えて，他のメトリクスの組み合わせよりも欠陥の発生を予測に有用であること [36] が確認されていることが挙げられる．

プログラムスライシングの結果を利用したメトリクスがいくつか提案されている [37]．例えば，メトリクス Tightness[37] は，C 言語における関数中の文のうち，全てのスライスに共通して含まれる文の割合であり，ほとんど文が返値や大域変数の値に影響与えていると高い値になる．直観的には，単一の目的で作成された関数は Tightness の値が高くなる．このようなプログラムスライシングに基づくメトリクスを用いることで，オープンソースソフトウェアに含まれる関数の凝集性が低下していることを定量化できることが確認されている [38]．Kataoka らは，リファクタリングの効果を計測する 3 つのメトリクスを提案している [39]．リファクタリング [40, 41] とは，保守性の改善を目的とした変更作業のことである．これらメトリクスは，メソッド間の結合に基づいてリファクタリングの効果を計測する．具体的には，1 つ目は返値を介した結合，2 つ目は引数を介した結合，3 つ目は変数の共有に基づく結合を計測する．リファクタリングを行う開発者は，これらメトリクスを用いることで，リファクタリングによりメソッド間に存在する結合がどのように変化したかを調査できる．

このように，ソフトウェア保守の技術は多数存在する．本論文では，昨今のコンピュータの処理能力の向上を受けて，解析可能な事象が増えてきたことでさかんに研究されているソースコード解析技術に注目し，ソフトウェア保守の支援をおこなう．

### 1.4.2　ソースコード解析

ソースコード解析は，静的解析と動的解析に分類できる．まず，本節ではソースコード静的解析とソースコード動的解析の各々について説明し，その関係について考察する．

**ソースコード静的解析**　ソースコードを実際に動作することなく解析することで，ソースコードからその性質や振る舞いを抽出し，それを開発者に提供する技術である．ソースコードの中身を扱うため，網羅性の高い解析ができる．

**ソースコード動的解析**　ソースコードを実際もしくは仮想的な動作環境上で実際にテ

ストケースを与えてプログラムを動作させ，その動作結果や動作中のログなどを解析することで性質や振る舞いに関する情報を開発者に提供する技術である．マルチスレッド処理など，ソースコード静的解析で発見しづらい振る舞いを解析できるが，網羅的な振る舞いを調べるには大量のテストケースが必要となる．

ソースコード静的解析とソースコード動的解析は解析しやすい性質や振る舞いが異なるため，お互いに補完する技術であるといえる．ただし，ソースコード動的解析はソースコードを実際に動作させる必要があるため，開発途中や修正途中の未完全なソースコードに対して利用できない．さらに，ソースコード動的解析するためには解析目的に則した十分な量のテストケースを準備する必要があるため，適用に対する初期コストがソースコード静的解析に比べて大きい．そこで，本論文では実際のソフトウェア保守の現場で適用しやすいソースコード静的解析に注目する．

ソースコード静的解析におけるグラフ化は，解析対象に存在する個々の部品間の関係を抽出し，抽象化し表現することを目的とした場合が多い．グラフ化の代表的な例として，以下を挙げる．

- プログラムのソースコードを木構造で表現した抽象構文木
- 手続き，メソッドなどの呼び出し関係をグラフ化した CFG(Call Flow Graph)
- クラスの継承関係をグラフ化したクラス階層構造
- プログラム間のデータフローや制御構造をグラフ化したプログラム依存グラフ

グラフ化によって個々の部品間の関係が明確になるため，構文木からプログラム依存グラフを作成する場合のように，グラフ化された情報を用いてより高度なプログラム解析が行われることも多い．例としては，エイリアス関係（同一メモリ空間を指す可能性のある式間の同値関係）にある変数の対の情報をもとに，より正確なデータフロー関係の解析などが挙げられる．一般的に複数の解析を組み合わせた場合，解析コストが上がるが，得られる情報の精度の向上が知られている．グラフの解析方法の例として代表的なものを以下に挙げる．

- グラフ上の辺の探索による，到達可能な節点の計算
- グラフの比較による，同一部分の検出
- クラスの階層構造の深さ，手続き（メソッド）の数などの数値化
- 利用関係などの関係の行列化

一方で，メトリクス計測を用いたソースコード解析は，解析対象における個々を抽象化し個々の性質を取り出すことを目的としている．メトリクスを用いた解析の代表的な例として以下を挙げる．

- クラス数，メソッド数，コード行数（LOC）などのメトリクス
- トークンの抽象化を目的とした記号化

- プログラムの品質や再利用性の評価値
- ソフトウェア間の類似性

メトリクスとして計測された情報は個々の性質をある観点から観測したもので，これらの情報を複数組み合わせることで，より多面的な観点から個々の解析対象を観測できる．そのため，これらの数値を組み合わせることで，部品を評価するための新たな評価基準を生み出すができることも多い．数値化された情報の多くは，統計的手法を用いた評価に利用されることが多い．また，記号化された情報は個々の性質をある観点から観測したものであるが，配列化や行列化を行うことで，解析対象全体の特徴を示すことができる．そのため，統計的手法を用いた評価が行われることもあるが，単に比較するために利用されることも多い．

### 1.4.3 ソースコード静的解析によるソフトウェア保守支援

ソースコード静的解析技術を利用してプログラムから抽出された情報をもとに，ソフトウェア保守の支援を目的として様々な解析が行われている．グラフ化した情報を用いたソースコード静的解析の代表的な例を以下に示す．

**最適化コードの生成**　コンパイル時に必要のない命令を削除する
**テストデータの自動生成**　テストを行いたい実行経路を通るような入力データを実行履歴から生成する
**プログラムの結合**　似た部分を結合することで，ただ単に結合した場合よりも高速化を計る
**デバッグ支援**　プログラムスライス [37] を用いることで，デバッグ対象を限定する
**影響波及解析**　再テストすべきテストケースを限定することで，テスト工程を効率化する
**モデルチェック**　プログラムの正当性や安全性の検証
**情報漏洩解析**　プログラムの中で，セキュリティポリシーを満たさない文を検出する
**プログラム理解支援**　解析結果情報を提示することで，保守およびデバッグ作業を支援する

次に，メトリクス計測を用いたソースコード静的解析の代表的な例を以下に示す．

**ソフトウェア部品の評価**　メトリクス値化された部品の性質から再利用性や品質を評価
**コードクローンの把握**　コピーされたソースコードの検出する
**コピー部品の把握**　メトリクス計測された情報を配列化し，解析効率を上げる
**ソフトウェア（部品）のクラスタリング**　メトリクス値等を比較し，同じ傾向にあるソフトウェア (部品) を分類する

**理解支援**　解析結果情報を選別の基準とし，大量の部品からの選別作業を支援する

なお，ここで挙げる例は一部で，抽出されるプログラム解析情報および利用目的はこれら以外にも多く存在する．さらに，ここで挙げたいくつかのプログラム解析情報を組み合わせることで，新たな解析をする手法も考案されている．

## 1.5　本研究の概要

1.3 節では，ソフトウェア保守とモダナイゼーションの関係について述べた．本研究では，運用フェーズで繰り返し行われるソフトウェア保守だけでなく，広範囲な変更を伴うモダナイゼーションも含めたソフトウェア保守全般を研究対象とする．具体的には，ソフトウェア保守およびモダナイゼーションにおいて発生する以下の課題について，ソースコード静的解析を用いて解決を試みる．

**ソフトウェア保守におけるコードクローンの把握**

- 情報検索技術に基づく細粒度ブロッククローン検出
- 情報検索技術と深層学習を用いたコード片類似性判定法の比較調査

**モダナイゼーションにおける費用対効果の見積り**

- 段階的再構築における依存関係分析を用いた費用対効果の試算

### 1.5.1　ソフトウェア保守におけるコードクローンの把握

1.1 節でも述べたとおり，ソフトウェアの保守における問題のひとつとして，コードクローンが指摘されている [8, 9]．コードクローンに対する様々な保守や管理の方法が提案されているが，ソースコードの規模が大きくなるとソースコード中に含まれるコードクローンも膨大な量となり，手作業でそれらを管理することは困難となる．そこで，コードクローンを自動的に検出することを目的とした様々なコードクローン検出手法が提案されている [8, 42].

Roy らはコードクローンの定義として，コードクローン間の違いの度合いに基づき以下の 4 つのタイプに分類している [43].

**タイプ 1**　空白やタブの有無，コーディングスタイル，コメントの有無などの違いを除き完全に一致するコードクローン．

**タイプ 2**　タイプ 1 の違いに加えて，変数名などのユーザー定義名，変数の型などが異なるコードクローン．

**タイプ 3**　タイプ 2 の違いに加えて，文の挿入や削除，変更などが行われているコードクローン．

**タイプ 4**　類似した処理を実行するが，構文上の実装が異なるコードクローン

タイプ 4 のコードクローンは，構文的に類似していないが処理内容の意味的に類似したコード片であり，具体的には以下のような差異が挙げられる．

- 条件分岐処理や繰り返し処理などの制御構造の実装が異なる．
- 中間媒介変数の利用の有無が存在している．
- 文の並び替えが発生している．

構文的に類似していないタイプ 4 のコードクローンは，全く異なる書き方がされていることが多い．そのため，既存のコードクローン検出法の多くはタイプ 4 のクローン検出が難しい [15, 16]．

本研究では，情報検索という技術を活用し，タイプ 1 からタイプ 4 までのコードクローンを検出する手法について研究する．情報検索とは，コンピュータを用いて大量のデータ群から目的に合致した情報を取り出す技術であり，自然言語で記述された文書のみならず，ソースコードを含む多様な情報を対象とすることが可能である [44]．情報検索の分野において，ベクトル空間モデルを用いて意味的に類似した文書を検索する手法が広く知られている．ベクトル空間モデルを用いる手法では，文書を多次元の特徴ベクトルで表現し，文書間の意味的な類似性の判定をベクトル間の類似度計算に帰着させる [45]．このベクトル空間モデルで使用されるベクトル表現を，情報検索技術に基づくベクトル表現と呼ぶ．本研究では，ソースコードを情報検索技術に基づくベクトル表現へと変換し，ベクトル表現間の類似度を計算することでコードクローンを検出する．この手法は，構文的類似性に限らず，意味的類似性も捉えられる可能性があるため，タイプ 1 からタイプ 4 までの幅広いコードクローンの検出が期待できる．

そこで本研究では，ソフトウェア保守におけるコードクローンの把握に関する以下の 2 つの研究課題に取り組む．

**■情報検索技術に基づく細粒度ブロッククローン検出**　最初に，情報検索技術に基づく細粒度ブロッククローン（コードブロック単位のクローン）の検出手法を提案する．既存研究として，山中らは情報検索技術の一種である TF-IDF[44] を利用し，関数単位でコードクローンを検出する手法（関数クローン検出法）を提案した [46]．関数クローン検出法はタイプ 1 からタイプ 4 までのコードクローンを検出できる一方で，検出粒度が大きいために検出漏れが起きるという課題があった．特に長年保守されたソフトウェアの場合，1 つの関数内で多様な処理が実装されていることが多く，関数単位の検出ではコードクローンを検出できないことがある．このような検出漏れを減らすために，本研究では関数単位よりも検出粒度の細かいブロッククローン（コードブロック単位のクローン）を検出する手法を提案する．本手法では，構文解析によりコードブロックを抽出し，TF-IDF を利用してコードブロックをベクトル表現に変換し，ベクトル表現間のコサイン類似度を計算することによって，タイプ 1 からタイプ 4 のブロッククローンを検出する．また，クラスタリングの手法やベクトル表現のデータ構

造を工夫することにより，高速かつ低メモリ使用量の検出法を目指す．評価実験では，提案手法の精度および検出速度を評価し，既存手法と比較する．

**■情報検索技術と深層学習を用いたコード片類似性判定法の比較調査**　次に，情報検索技術と深層学習を用いたコード片の類似性判定法について調査する．コード片の類似性判定法はソフトウェア工学における重要な基礎技術であり，コードクローン検出やコード片検索などで使用される．コード片の類似性判定法では，構文的な類似性だけでなく，処理内容の意味的な類似性も判定することが重要である．前述した，TF-IDF を利用したベクトル表現とコサイン類似度を組み合わせた類似性判定法では，処理内容の意味的な類似性を高速に判定できる一方で，検出漏れが多いという課題があった．また一方で，深層学習を用いた類似性判定法も提案されているが，この手法は意味的な類似性を高い精度で判定できる反面，実行速度が遅いという課題があった．そこで本研究では，判定精度と実行速度の 2 つの観点から，情報検索技術に基づくベクトル表現と深層学習の効果的な組み合わせについて調査する．調査実験では，5 種類のベクトル表現間での比較と，深層学習を用いた既存手法との比較をする．

### 1.5.2　モダナイゼーションにおける費用対効果の見積りの難しさ

ビジネス上の変化への迅速な対応や保守開発工数の削減の観点から，レガシーシステムのモダナイゼーションに対する企業の需要は大きい [18]．特に，企業でモダナイゼーションを実施するシステムはビジネス上重要である場合が多いため，失敗リスクを最小限に抑えることが求められる．この失敗リスクを軽減する戦略として，システムの一部を切り出す段階的再構築が提案されている [47, 48]．

しかし，レガシーシステムのモダナイゼーションに対する企業の需要は大きい一方で，1.2 節でも述べたとおり，モダナイゼーションの費用対効果の見積りの難しさが指摘されている [18]．企業の経営陣は短期的な投資収益率を重視する傾向があり，一度投資が決定されると，その投資回収が優先される [18]．そのため，企業においては要件定義の前工程で行われるシステム化計画 [49] において，モダナイゼーションの費用対効果を見積もり，ユーザー企業とベンダー企業の間で合意を形成することが重要である [50]．特に，システム化計画工程において行われる見積りは，試算とよばれる [50]．

モダナイゼーションの失敗リスクを軽減する段階的再構築においても，費用対効果の試算は難しい．実際，著者が所属する企業で段階的再構築を実施した事例では，計画当初の見積りと実績値に乖離が生じている．そこで本研究では，モダナイゼーションにおける費用対効果の見積りに関する以下の研究課題に取り組む．

**■段階的再構築における依存関係分析を用いた費用対効果の試算**　本研究では，システムの依存関係分析を用いて段階的再構築の費用対効果を試算する手法を調査する．具体的には，過去に段階的再構築を実施した大規模な金融システムを対象にこの手法

を適用し，試算の妥当性を検証する．本研究では，システムの規模と依存関係の情報を活用し，費用として段階的再構築に必要な工数を，効果として削減可能な保守開発工数を試算し，実績値との乖離を比較する．また，プロジェクトの関係者にヒアリングを行い，適用手法の妥当性を定性評価する．

## 1.6 各章の構成

以降，第 2 章では情報検索技術に基づいて細粒度ブロッククローンを検出する手法について述べる．第 3 章では情報検索技術と深層学習を用いてコード片の類似性を判定する手法の比較調査について述べる．第 4 章では段階的再構築における依存関係分析を用いた費用対効果の試算を産業システムに適用し調査した結果について述べる．最後に，第 5 章では本論文のまとめと将来の研究方針について述べる．

# 第 2 章

# 情報検索技術に基づく細粒度ブロッククローン検出

## 2.1 まえがき

ソフトウェアの保守における問題のひとつとして，コードクローンが指摘されている [51]．コードクローンとは，ソースコード中に含まれる互いに一致または類似した部分を持つコード片のことであり，一般的に，コードクローンの存在はソフトウェアの保守を困難にすると言われている．コードクローンに対する様々な保守や管理の方法が提案されているが，ソースコードの規模が大きくなるとソースコード中に含まれるコードクローンも膨大な量となり，手作業でそれらを管理することは困難となる．そこで，コードクローンを自動的に検出することを目的とした様々なコードクローン検出手法が提案されている [8, 42]．

山中らは TF-IDF[44] と LSH[52] を利用することによって，意味的に処理が類似した関数単位のコードクローンを検出する手法（関数クローン検出法）を提案した [46]．コード片単位で検出を行う場合，構文の不完全な部分で終了するコード片など，集約を行うことが困難なコードクローンが多く検出されることがある [53]．一方，関数クローンは処理の内容がまとまっているため，開発者にとって集約の対象になりやすいコードクローンを検出できる．山中らの手法では，情報検索技術の一種である TF-IDF[44] を用いて，ソースコード中の識別子や予約語に利用される単語に対して重み付けを行う．そして，重み付けに基づいて各関数を特徴ベクトルに変換し，特徴ベクトル間の類似度を計算することによって，関数クローンの検出を行う．また，近似最近傍探索アルゴリズムの一種である LSH（Locality-Sensitive Hashing）[52] を用いて，特徴ベクトルをクラスタリングすることにより，検出の高速化を行っている．しかし，山中らの手法では検出粒度が関数単位のため，長い関数内の一部にコードクローンが含まれた場合，検出漏れが生じる可能性がある．このような検出漏れを減らすためには，関数単位より小さい粒度で構文上のまとまりがあるコードクローンの検

出を行うべきである．

そこで，本研究ではコードブロック単位のコードクローン（ブロッククローン）を検出する手法を提案する．関数単位より小さい粒度であるコードブロック単位で検出を行うことで，関数単位では検出できなかったようなコードクローンが検出でき，検出精度が向上する．本研究ではコードブロックを，関数と，関数内部の if，for，while 文などの中括弧で囲まれた部分と定義する．本手法では，まずソースコードに対して構文解析を行い，コードブロックの抽出を行う．その後，抽出した各コードブロックに対して TF-IDF を用いて特徴ベクトルに変換し，特徴ベクトル間の類似度を計算することでブロッククローンの検出を行う．また本手法でも山中らの手法と同様，LSH を用いた特徴ベクトルのクラスタリングを行い，検出の高速化を行う．

しかし，関数単位より小さい粒度で検出を行うため，検出時間とメモリ使用量が増大する問題が発生する．この問題に対応するため，以下の 2 点の工夫を行った．1 点目は，クラスタリング手法の LSH の変更である．LSH は様々な応用手法が研究されており，本研究では提案手法に適した LSH の検討を行った．その結果，multi-probe LSH [54] と cross-polytope LSH [55, 56] を組み合わせた手法 [55] を用いて特徴ベクトルのクラスタリングを行うことを選択した．multi-probe LSH は，従来の LSH が抱えるメモリ使用量の問題点を改良したアルゴリズムである [57]．また，cross-polytope LSH は TF-IDF を用いた特徴ベクトルのクラスタリングに適したアルゴリズムである [55]．これらを組み合わせた手法を用いることで少ないメモリで高速なクラスタリングが可能となった．2 点目は，特徴ベクトルを表現するデータ構造の変更である．TF-IDF を用いた特徴ベクトルが疎（ほとんどの要素が 0）である性質に着目し，0 以外の要素のみを保持するデータ構造として実装した．これにより，メモリ使用量や入出力時間の削減を実現した．上記の工夫を行うことで，山中らの関数クローン検出手法より検出粒度が小さいにもかかわらず，より高速な検出を実現し，さらに大規模な検出対象に対しても適用可能となった．

評価実験では，関数クローン検出法 [46] と，字句単位のコードクローン検出ツールである CCFinderX [58] の，2 つの手法を検出精度と検出時間の観点から比較を行った．3 つの C 言語のプロジェクトに対して適用した結果，本手法が総合的に高い精度でより多くのコードクローンを検出することができた．また，本手法は関数クローン検出法や CCFinderX と比較して高速にコードクローンを検出することができた．また，スケーラビリティの評価を行い，1KLOC から 10MLOC までの検出対象において線形的に検出時間が増加することを確認した．

以降，2.2 章では，本研究の背景として関数クローン検出法について述べる．2.3 章では，本研究で提案するブロッククローン検出法について述べる．2.4 章では，本手法の評価実験について述べる．2.5 章では，本研究の考察について述べる．2.6 章では，関連研究について述べる．最後に，2.7 章でまとめと今後の課題について述べる．

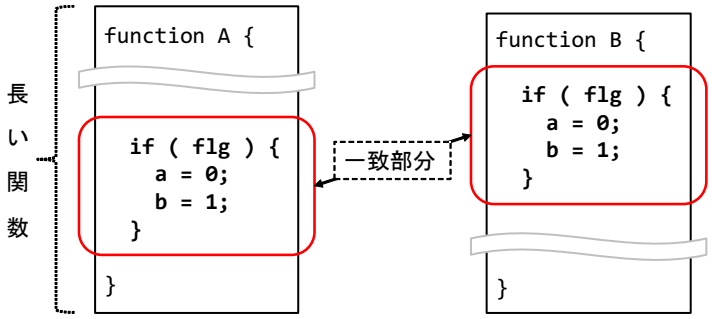


図 2.1　長い関数内の一部にコードクローンを含む例

## 2.2　コードクローン

本章では，本研究の背景として山中らの関数クローン検出法と，その問題点について述べる．

### 2.2.1　関数クローン検出法

山中らは TF-IDF と LSH を利用することによって，意味的に処理が類似した関数クローンを検出する手法を提案した [46]．コード片単位でコードクローンの検出を行う場合，構文の不完全な部分で終了するコード片など，集約を行うことが困難であるコードクローンが多く検出される恐れがある [53]．一方，関数クローンは処理の内容がまとまっているため，開発者にとって集約の対象になりやすいコードクローンが検出できる．また関数クローン検出法は，1.5.1 項で説明したタイプ 1 からタイプ 4 までのコードクローンを検出可能である．この手法は，まず，入力されたソースコード中のワード（例：予約語，識別子名）に基づいて各関数を特徴ベクトルに変換する．そして，特徴ベクトル間の類似度を求めることによってクローンペア（互いに処理が類似したクローンの対）の集合をリストとして出力する．また，類似度の計算の直前に LSH [52] を利用し，特徴ベクトルのクラスタリングを行うことによって，検出の高速化を行っている．

### 2.2.2　関数クローン検出法の問題点

関数クローン検出法に対して以下の 2 つの問題点が挙げられる．

1 つ目は，関数単位の検出による問題点である．この手法は，関数全体ではなく，一部のみがコードクローンになっている場合に検出することができない．例えば図 2.1 のように，長い関数内の一部にコードクローンが含まれる場合，検出漏れが生じる可能性がある．

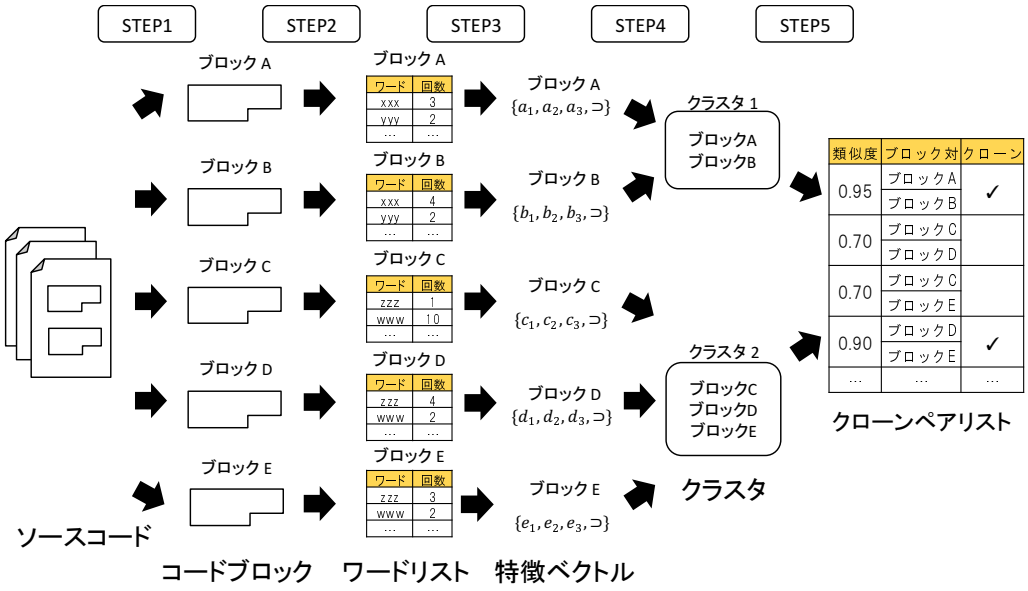


図 2.2　提案手法の概要

2 つ目は，メモリ使用量が多いという問題点である．関数を特徴ベクトルに変換する方法として関数クローン検出法は TF-IDF を用いているが，TF-IDF では全関数で出現したワードの種類数が次元数となるため，特徴ベクトルの次元数が非常に大きくなる傾向にある．特徴ベクトルは各関数に 1 個ずつ与えるため，次元数の大きい特徴ベクトルを多数保持することになり，メモリ使用量が大きくなる．また，高速に検出を行うために LSH を用いてクラスタリングを行っているが，LSH は精度を上げるとメモリ使用量が大きくなる特徴がある．よって，大規模プロジェクト（Linux Kernel など）に対してコードクローンの検出を行う場合，メモリ不足で検出を完了できない恐れがある．

## 2.3　提案する検出手法

2.2.2 節で述べた 2 つの問題点を踏まえ，新たなコードクローン検出法の必要性が考えられる．そこで，本研究では関数単位より検出粒度を小さくした，コードブロック単位のコードクローン検出法（ブロッククローン検出法）を提案する．ブロッククローン検出法では，関数クローン単位では検出できなかったコードクローンの検出が可能になる．例えば図 2.1 のように，長い関数内の一部にコードクローンが含まれる場合も，提案手法では検出が可能である．また，字句単位の検出より処理の内容がまとまっているため，開発者にとって集約などの保守作業の対象となりやすい．以上の観点から本手法には有用性があると考えられる．

本研究では，2.2.1 節で説明した山中らの関数クローン検出法を基に，タイプ 1 か

らタイプ 4 すべてに対応したブロッククローン検出法を提案する．本手法の概要を図 2.2 に示す．本手法は主に以下の 5 つのステップで実行される．

STEP 1　ソースコードから抽象構文木を生成し，生成した抽象構文木からコードブロック（2.3.1 節参照）を取り出す．

STEP 2　STEP 1 で抽出した各コードブロックから，ワード（2.3.1 節参照）の抽出を行う．

STEP 3　TF-IDF を利用し，STEP 2 で抽出したワードに重み付けを行い，各コードブロックを特徴ベクトルに変換する．

STEP 4　LSH を利用し，STEP 3 で求めた各コードブロックに対する特徴ベクトルのクラスタリングを行う．

STEP 5　STEP 4 で求めたコードブロックの各クラスタの中で，特徴ベクトル間の類似度の計算を行い，ブロッククローン（2.3.1 節参照）を検出する．

本手法と関数クローン検出法の相違点は，以下の 3 点である．

- コードクローンの検出粒度
- 特徴ベクトルをクラスタリングする手法の選択と実装
- 特徴ベクトルを表現するデータ構造の選択と実装

主な相違点はコードクローンの検出粒度を小さくした点である．本手法では関数単位だけでなく，関数内のコードブロック単位の両方のコードクローンを検出する．検出粒度を関数より小さくすることで，検出精度の向上を実現した（2.4.1 節参照）．しかし，検出粒度を小さくすると検出対象数が増加し，それに伴い検出時間とメモリ使用量が増大する問題が新たに発生する．この問題に対処するため，さらに 2 つの変更を適用した．

まず，特徴ベクトルのクラスタリングを行う LSH を変更した．関数クローン検出法でも LSH を用いて特徴ベクトルのクラスタリングを行われており，これにより高速な検出を実現している．しかし，近年 LSH は様々な応用手法が研究されており，関数クローン検出法では LSH の応用手法の検討まではされていない．また, LSH によるクラスタリングが占める時間の割合が大きく無視できないため [59]，本手法に適した LSH の検討を行った．その結果，multi-probe LSH [54] と cross-polytope LSH [55, 56] の 2 つの既存手法を組み合わせる手法 [55] に変更した．multi-probe LSH とは，空間計算量の改良を行った LSH である．また，cross-polytope LSH とは，角度距離に基づいてハッシュ化を行う LSH である．角度距離に基づいた cross-polytope LSH は TF-IDF を用いた特徴ベクトルのクラスタリングが高速に行える．multi-probe LSH を cross-polytope LSH に組み合わせることで，より少ない空間計算量でより高速にクラスタリングが可能なることが示されている [55].

さらに，特徴ベクトルを表現するデータ構造を変更した．特徴ベクトルは，検出対

象数の増加に伴いベクトルの次元が高次元になる性質があり，次元が高次元になるほどメモリ使用量とベクトルの計算時間が増大する．関数クローン検出法では，特徴ベクトルのすべての要素の値を保持するデータ構造をしている．関数単位の検出では問題がないが，検出粒度を小さくしコードブロック単位で検出を行うには，上記の高次元の問題が無視できなくなる．そこで，本手法のように TF-IDF を用いた特徴ベクトルでは，ほとんどの要素が 0 である疎なベクトルとなる性質に着目し，非 0 要素のみを保持するデータ構造を選択した．データ構造を疎なベクトルとして実装することで，メモリ使用量や計算時間，さらに入出力時間の削減を行った．

以上の変更により，関数クローン検出法と比較して検出粒度は小さくなったにもかかわらず，より高速な検出を実現し，さらに大規模な検出対象に対しても適用可能となった．

### 2.3.1 用語の定義

**コードブロック**

本研究では，プログラミング言語において，複数の命令文を一括りにまとめたものをコードブロックという．多くのプログラミング言語では，コードブロックを入れ子構造にすることができ，変数のスコープとしての意味を持つことがある．

本手法では，以下の 2 つの条件のいずれかを満たすコードブロックを検出対象とする．対象言語は C 言語と Java 言語とする．

**条件** 1-1　関数の ‘{ }’ で囲まれた範囲
**条件** 1-2　if，else，for，while，do-while，switch 文の ‘{ }’ で囲まれた範囲

ただし，後に ‘{ }’ が現れない単文の命令文はコードブロックとしての纏まりがないため検出対象としない．また図 2.3 の Block A に対する Block B や Block C のように，入れ子構造における親を持つコードブロックを検出可能であり，検出対象を再帰的に探索する．

**ワード**

本手法では以下の条件 2-1，2-2 のいずれかを満たすものをワードとして定義する．

**条件** 2-1　予約語
**条件** 2-2　識別子名を構成する単語

識別子名が複数の単語から構成される場合，以下の方法でワード単位に分割する．

- ハイフンやアンダースコアなどの区切り記号による分割
- 識別子名中の大文字になっているアルファベットによる分割

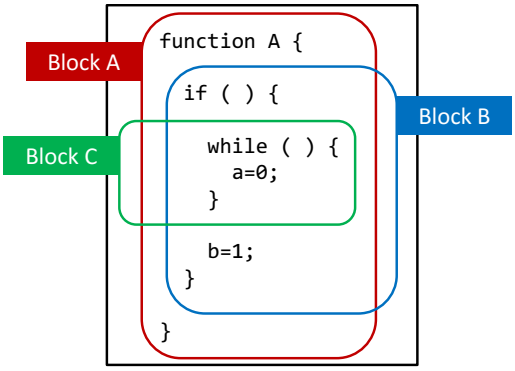


図 2.3　入れ子構造において親子関係にあるコードブロック

また，2 文字以下の識別子は，それらをまとめて同一のメタワードとして扱う．例えば，繰り返し文などでよく利用される i や j といった変数は，意味情報が込められていない変数として扱うためである．さらに，条件分岐に用いられる if や while，繰り返しに用いられる for や while などの予約語もワードとして扱う．なお，各ワードの大文字と小文字による区別はつけず，同一のワードとして扱う．

**ブロッククローン**

本手法におけるブロッククローンについて説明する．最初にブロッククローンペアについて定義する．本手法では，以下の条件 3-1，3-2 の両方を満たすコードブロック対を，ブロッククローンペアと呼ぶ．

**条件** 3-1　コードブロック間の類似度が閾値以上

$$sim(CB_1, CB_2) \geq p \quad (0 \leq p \leq 1)$$

**条件** 3-2　入れ子構造において，$CB_1$ が $CB_2$ の親でなく，かつ $CB_2$ が $CB_1$ の親でない

次に極大ブロッククローンについて定義する．$CB_1$，$CB_2$ がブロッククローンペアであり，かつ $CB_1$，$CB_2$ それぞれを真に包含する如何なるコードブロックもブロッククローンペアでないとき，$CB_1$，$CB_2$ を極大ブロッククローンと呼ぶ．本手法では，極大ブロッククローンをブロッククローンと定義する．条件 3-2 で示したように，ブロッククローンペアはコードブロック間に入れ子構造における親子関係がないことが条件である．コードブロック間に入れ子構造における親子関係が存在する場合，一方のコードブロックが他方を包含していることを示している．例えば，図 2.3 のコードブロック A と B は親子関係が存在し，包含関係にあるためブロッククローンペアでない．

また，極大ブロッククローンをブロッククローンと定義するとは，言い換えるとそれぞれ入れ子関係にあるコードブロックの類似度が閾値以上の場合，最も外側のコー

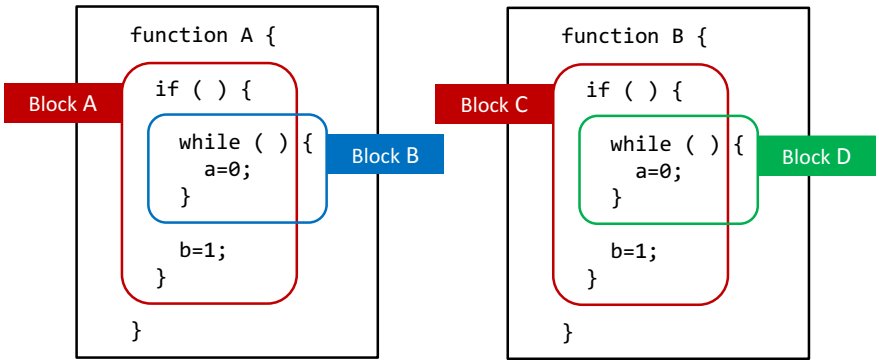


図 2.4　極大コードブロックペアと重複したコードブロックペア

ドブロックペアをブロッククローンとするという意味である．例えば，図 2.4 のコードブロック A と C，B と D それぞれの類似度が閾値以上となる場合，最も外側のコードブロック A と C をブロッククローンとする．

### 2.3.2　コードブロックとワードの抽出

本手法でのコードブロックとワードの抽出について説明する．最初にソースコードに対して構文解析を行い，抽象構文木を生成する．本手法では，構文解析には ANTLR v4[*1]を利用する．次に生成した抽象構文木に対して深さ優先探索で検索する．抽象構文木から関数（2.3.1 節の条件 1-1 を満たすコードブロック）の部分木を取り出し，コードブロックとして抽出する．そして，取り出した部分木から，コードブロック（2.3.1 節の条件 1-2 を満たすコードブロック）の部分木を取り出し，関数の内側に存在するコードブロックを抽出する．コードブロックの抽出後，各コードブロック内に含まれるワードの抽出を行う．

### 2.3.3　特徴ベクトルの計算

特徴ベクトルの計算では，ワードに対し TF-IDF[44] を利用して重みを計算し，その値を特徴量として各コードブロックを特徴ベクトルに変換する．よって，各コードブロックの特徴ベクトルの次元数はソースコード中に存在する全ワードの種類数となる．本手法では，tf 値はコードブロック中のワードの出現頻度を，idf 値はソースコード中のワードの希少さを表している．tf 値は式 (2.1)，idf 値は式 (2.2) で与えられる．

$$tf_{i,j} = \frac{n_{i,j}}{\sum_{k \in b_j} n_{k,j}} \tag{2.1}$$

[*1] http://www.antlr.org/

$$idf_i = \log \frac{|F|}{|\{f : f \ni w_i\}|} \tag{2.2}$$

ここでは，$n_{i,j}$ はコードブロック $b_j$ 内におけるワード $w_i$ の出現回数，$\sum_{k \in b_j} n_{k,j}$ はコードブロック $b_j$ における全ワードの出現回数の和, $|F|$ は全関数の数, $|\{f : f \ni w_i\}|$ はワード $w_i$ が出現する関数の数を示している．

関数クローン検出法と比較して，tf 値の求め方をコードブロック単位に変更したが，idf 値の求め方はコードブロック単位に変更せず関数単位のままである．なぜなら，コードブロック単位で idf 値を求めるとワードの重み付けに偏りが生じてしまうからである．あるワードがいくつのコードブロックに含まれるかは出現場所によって異なる．例えば，図 2.3 の関数内の変数 a はブロック A と B の 2 個のコードブロックに含まれるが，変数 b はブロック A のみにしか含まれない．そのため，ソースコード中の出現回数は同じにもかかわらず，出現するコードブロックの数が異なってしまう．idf 値はワードの希少性に基づいて重み付けを行っているため，偏りを無くすために関数単位で求めた．

また，TF-IDF を用いた特徴ベクトルは，非常に高次元かつ疎となる特性がある．そのため，非 0 要素のみを保持する疎ベクトルとして実装することでメモリ使用量の効率を改善した．さらに，特徴ベクトルの外部記憶装置への入出力処理の時間を高速化した．

### 2.3.4 特徴ベクトルのクラスタリング

特徴ベクトルのクラスタリングとして，近似最近傍探索の一種である LSH[52] を用いた．LSH とは，高次元なデータを確率的にハッシュ化し，近似的に近傍点を見つけるアルゴリズムである．クローンペアとなる特徴ベクトルは近傍点で表されるため，LSH を用いてクラスタリングすることで類似した特徴ベクトルを高速に絞り込むことができ，クローン検出の高速化を実現している．

最初に，LSH の一般的な説明を述べる．LSH は，$(c, r)$-Approximate Nearest Neighbor(ANN) と，$(r, cr, p_1, p_2)$-sensitive hash を用いて表される．$(c, r)$-ANN とは，近似最近傍探索の問題定義であり，$(r, cr, p_1, p_2)$-sensitive hash とは，空間的に距離が近い点が同じハッシュ値を取る確率が高くなる確率的ハッシュ関数である．$(c, r)$-ANN と，$(r, cr, p_1, p_2)$-sensitive hash の定義を以下に示す．

$R^d$ 上に距離関数 $D$ を定義する．

$(c, r)$-ANN　実数 $c > 1$, 実数 $r > 0$, 任意のクエリ $q \in R^d$ が与えられたとき, $p \in R^d$ を $q$ の正解最近傍点とし，$D(p, q) < r$ ならば，

$$P' = \left\{ p' \in R^d : D(p', q) < cr \right\}$$

で定義される $R^d$ の部分集合 $P'$ を求める問題を $(c, r)$-ANN と呼ぶ．

$(r, cr, p_1, p_2)$-sensitive hash　任意の点 $p$，$q \in R^d$ が与えられたとき，

- if $D(p,q) < r$ then $\Pr[h(p) = h(q)] \geq p_1$
- if $D(p,q) > cr$ then $\Pr[h(p) = h(q)] \leq p_2$

を満たすハッシュ関数 $h$ を $(r, cr, p_1, p_2)$-sensitive hash と呼ぶ．Pr は条件式が真となる確率を表している．

ここでは一般的な LSH で用いられる定義を示したが，上記のハッシュ関数 $h$ を変更するなど様々な LSH の応用手法が提案されている．関数クローン検出法ではユークリッド空間に対する LSH を実装した E2LSH[60][*2]を用いてクラスタリングを行い検出の高速化を実現しているが，他の LSH の検討までは行われていない [46]．そこで本研究では，本手法により適した LSH の検討を行った．

一般的に LSH を用いて大規模のデータセットを高い精度で求めようとすると，メモリ使用量が非常に大きくなるという問題点がある [57, 54]．そこで，メモリ使用量を改良するため multi-probe LSH[54] という手法が提案されている．LSH のアルゴリズムは，空間的に距離が近い 2 点が同じハッシュ値になる確率が高くなるようなハッシュ関数を用い，同じハッシュ値を取る点を同じバケットに入れることで近傍点の検索を行う．しかし LSH は確率的手法であるため，近い 2 点が偶然に別のバケットに入る可能性が存在する．従来の LSH では複数のハッシュテーブルを用いることで確率的な誤差を少なくし精度を上げている．そのため，大規模なデータセットに対してはより多数のハッシュテーブルが必要となり，LSH のメモリ使用量の増大につながっている．multi-probe LSH は，ある点が入るバケットだけでなく，空間的に距離が近いバケット群も調べるという手法である．これにより，少ないハッシュテーブルでも偶然別のバケットに入った点の見落としを防いでいる．実際に，192 次元のデータセットに対して同じ再現率 (0.90) を得るために，従来の LSH は 49 個のハッシュテーブルが必要だったのに対し，multi-probe LSH は 3 個のハッシュテーブルで検索時間をほぼ落とさずに達成したという結果が示されている [54].

本手法では，TF-IDF を用いた特徴ベクトルに対し，コサイン類似度によって類似したベクトルの検索を行うが (2.3.5 節参照)，これらは情報検索の分野でよく用いられる手法であり，上記のようなベクトル空間に対する LSH がいくつかある．その中で，角度距離に基づいてハッシュ化を行う cross-polytope LSH[55, 56] が，理論的に保証されており計算時間の観点から実用的な手法として提案されている．cross-polytope LSH は，実用性の面で優れているとして既に提案されていた hyperplane LSH[61] と比較し，1.2 倍から 4.0 倍高速に同じ精度で検索できるという結果が示されており，さらに TF-IDF を用いた特徴ベクトルの場合は少なくとも 3 倍高速になるという結果も示されている [55].

Andoni らは multi-probe LSH を cross-polytope LSH に組み合わせた手法 [55] を

[*2] http://www.mit.edu/ andoni/LSH/

提案しており，この 2 つの LSH を組み合わせた手法を本研究では選択した．なお，2 つの手法の組み合わせの実装として FALCONN [55][*3] ライブラリを利用した．

### 2.3.5 特徴ベクトルの類似度の計算

本手法ではコサイン類似度を用いてクローンペアの判定を行う．コサイン類似度は多次元ベクトルの類似度を表す尺度であり，次元が V である 2 つの特徴ベクトル $\vec{a}, \vec{b}$ 間の類似度は以下の式 (2.3) で与えられる．

$$sim(\vec{a}, \vec{b}) = \cos(\vec{a}, \vec{b}) = \frac{\sum_{i=1}^{|V|} a_i b_i}{\sqrt{\sum_{i=1}^{|V|} {a_i}^2}\sqrt{\sum_{i=1}^{|V|} {b_i}^2}} \tag{2.3}$$

TF-IDF の計算式より，特徴量は常に正の値を取るため，コサイン類似度は 0 から 1 の範囲となる．コサイン類似度が閾値以上であれば，それら 2 つのコードブロックはクローンペアであると判定する．閾値は実行時の引数によって与えられる．

## 2.4 評価実験

本章では，本研究で提案したブロッククローン検出法の評価実験について述べる．評価実験では，検出精度，検出時間，スケーラビリティの 3 つの観点から，本手法と既存手法の比較を行い評価した．また，保守対象とならないコードクローンと，コードクローンに対する保守作業の調査も行った．最後にブロッククローンの実例を示し，本手法の特徴について考察する．

2.4.1 節の本手法と既存手法の検出精度の評価では，本手法の拡張元であり検出粒度が異なる関数クローン検出法と，高速かつスケーラビリティの高い字句単位の検出法の中で代表的である CCFinderX [58] を比較対象として選び評価した．2.4.4 節の検出時間とスケーラビリティの評価では，本手法の有用性を確認するため，上記の 2 つに加えて，抽象構文木を用いた検出法の Deckard[62]，プログラム依存グラフを用いた検出法の Scorpio [63, 64] の 2 つを比較対象として追加して評価した．さらに 2.4.7 節では，同じコードブロック単位の検出手法として，粗粒度なコードクローン検出法 [65] を対象とした比較実験も行った．[*4] なおブロッククローンを検出する閾値は，本実験

[*3] https://falconn-lib.org/

[*4] 本実験において，本手法では最小一致トークン数を 30，類似度を 0.9 に，関数クローン検出法では最小一致トークン数を 30，類似度を 0.9 に，CCFinderX では最小一致トークン数を 50，最小トークンタイプ数は 12 に，Deckard では最小一致トークン数を 50，類似度を 0.85，トークンストライドを 2 に，Scorpio では最小一致ノード数を 7 に，粗粒度なコードクローン検出法では最小一致トークン数を 50 と設定した．本手法と関数クローン検出法の各パラメータは既存研究で信頼性が高いと示された値を採用した [46]．CCFinderX，Deckard の各パラメータは既存の実験で実際に用いられた値を採用した [66]．Scorpio のパラメータは既存研究を参考に設定した [64]．粗粒度なコードクローン検出法のパラメータは CCFinderX を参考に設定した．

では関数クローン検出法の論文に基づき，0.9 と設定し検出を行った [46].

2.4.1 節の評価実験においては，複数の手法の検出結果からクローンペアをサンプリングし，検出結果が正しくコードクローンであるか目視で判断するだけでなく，実際に保守対象となり得るかコードクローンの研究者にアンケートを行うことで検出精度を評価した．したがって，単にコード片が外見上類似しているだけでなく，ソフトウェアを保守管理するという観点から実際に有用なコードクローンの評価方法となっている．検出されたコードクローンがどの程度保守の対象になるのかは実際の開発で利用する際に非常に重要となる．既存の研究ではあまり評価されていなかったが，本研究ではこの面についても評価した．

このように，複数の手法の検出結果からクローンペアをサンプリングし，人手でコードクローンか否かを判断して精度を評価する方法は，コードクローン検出手法の比較評価で一般的に用いられる [67]．コードクローンには明確な定義がなく曖昧であるため，人間の判断が重要となる．また，膨大なコードクローンの検出結果をすべて人手で判断することは困難であるため，サンプリングにより判断対象を絞り込む必要がある．本研究では，ブロック単位，関数単位，字句単位といった検出粒度の異なる代表的なクローン検出ツールを選定し，これらの検出結果からクローンペアをサンプリングした．そのため，サンプリングされたクローンペアには，異なる粒度のクローンペアが偏りなく含まれている．

### 2.4.1 関数クローン検出法と CCFinderX との比較

本節では関数クローン検出法と CCFinderX の 2 つの既存手法との比較実験について述べる．コードクローンに対する保守作業として，集約や同時修正が挙げられる．そのため，今回は集約または同時修正の保守対象となるコードクローンを検出できるかという観点で検出精度を評価した．本実験では，対象プロジェクトから作成したベンチマークに対する検出精度と検出時間の観点から比較を行う．本実験で検出対象としたプロジェクトの一覧を表 2.1 に示す．

表 2.1　検出対象プロジェクト

| プロジェクト | バージョン | 言語 | 規模 |
|---|---|---|---|
| Apache HTTPD[*5] | 2.2.14 | C/C++ | 343 KLOC |
| PostgreSQL[*6] | 8.5.1 | C/C++ | 937 KLOC |
| Python[*7] | 2.5.1 | C/C++ | 435 KLOC |

[*5] http://httpd.apache.org/
[*6] http://www.postgresql.org/
[*7] http://www.python.org/

#### ベンチマークの作成方法

ベンチマークの作成は以下の 3 ステップで行った.

1. 表 2.1 のプロジェクトに対し，本手法，関数クローン検出法，CCFinderX の 3 つの手法でコードクローンを検出.
2. 各手法が各プロジェクトから検出したクローンペアから，30 個のクローンペアをランダムサンプリングし，合計 270 個のクローンペア集合を作成.
3. (2) で作成した 270 個のクローンペアに対し，目視で集約または同時修正の保守対象となるコードクローンかの判断を行い，ベンチマークを作成.

なお，ベンチマークに客観性を持たせるため，アンケートにより第三者にコードクローンの判断を依頼した．アンケートの概要を以下に示す.

**調査対象**　以下の 3 名に依頼

- コードクローンの研究者 1 名
- コードクローンの研究に従事している大学院生 2 名

**質問内容**　集約または同時修正の保守対象となるか

**回答方式**　二択（はい/いいえ）

上記の質問を，検出結果からサンプリングした 270 個のクローンペアに対して行った．そして，過半数である 2 人以上が保守対象と回答したクローンペアを正解とし，本実験で用いる正解クローンペア集合としてベンチマークを作成した．これはアンケートの結果が個人の判断に依存しないようにするためである．本実験では，各プロジェクトの正解クローンペア数は，Apache HTTPD が 74 個，PostgreSQL が 46 個，Python が 62 個となった.

#### 比較結果

本節では，関数クローン検出法と CCFinderX との比較実験の結果について述べる．本実験では，ベンチマークを用いた検出精度と検出時間の観点から比較を行った．検出精度の指標として，適合率，再現率，F 値の 3 つの指標を用いた．本実験における 3 つの指標を表 2.2 に示す．それぞれの手法と比較して，最も高い値を太字で示している.

#### 適合率

適合率とは，検出結果に対して真に正しかった割合を指し，正確性に関する指標として用いられる．本実験では，各手法が検出したクローンペアから，それぞれ 30 個ずつランダムサンプリングした．そして，サンプリングしたクローンペアについてアンケートを実施し，保守対象と判断された割合によって適合率を求めた．今回は過半数

である 2 人以上がコードクローンと判断した場合を正解としている．この適合率の求め方は，Bellon らの評価方法 [67] に基づいている．

本手法では, Apache HTTPD と Python において, 関数クローン検出法や CCFinderX より高い適合率が得られた．また，PostgreSQL においては，CCFinderX よりは高い適合率が得られたが，関数クローン検出法より低い適合率となった．3 つのすべてのプロジェクトの合計では適合率は 0.68 であり，関数クローン検出法と同程度の適合率，CCFinderX より高い適合率であることが確認できた．

本手法が PostgreSQL において適合率が低くなった原因として，出力処理の連続，同じ識別子の多用，2 文字以下の変数の多用，if 文の連続のクローンが多数あることが確認できた．

**再現率**

再現率とは，正解集合に対して実際に検出された割合を指し，網羅性に関する指標として用いられる．本実験では，アンケートによって作成したベンチマークの正解集合に対し，各手法が検出したクローンペアの割合によって再現率を求める．この再現率の求め方は，Bellon らの評価方法 [67] に基づいている．

本手法では，Apache HTTPD，PostgreSQL において，関数クローン検出法や CCFinderX より高い再現率が得られた．また，Python においては，関数クローン検出法よりは高い再現率が得られたが，CCFinderX より低い再現率となった．3 つのすべてのプロジェクトの合計では，再現率は 0.70 であり，関数クローン検出法と CCFinderX より再現率が高いことが確認できた．

表 2.2　検出精度の評価

| 検出手法 | 検出対象 | 適合率 | 再現率 | F 値 |
|---|---|---|---|---|
| 本手法 | Apache HTTPD | **0.90** | **0.74** | **0.81** |
| | PostgreSQL | 0.57 | **0.87** | 0.69 |
| | Python | **0.90** | 0.53 | 0.67 |
| | 合計 | **0.68** | **0.70** | **0.69** |
| 関数クローン検出法 | Apache HTTPD | 0.87 | 0.53 | 0.66 |
| | PostgreSQL | **0.83** | 0.74 | **0.78** |
| | Python | 0.30 | 0.21 | 0.25 |
| | 合計 | 0.67 | 0.47 | 0.55 |
| CCFinderX | Apache HTTPD | 0.70 | 0.55 | 0.62 |
| | PostgreSQL | 0.13 | 0.33 | 0.19 |
| | Python | 0.87 | **0.63** | **0.73** |
| | 合計 | 0.57 | 0.52 | 0.54 |

Python において再現率が低くなった原因として，Python にタイプ 2 のコードクローンが多く含まれることが確認できた（表 2.5 参照）．表 2.5 は，検出対象ごとのタイプ別の正解クローン数を示している．本手法は識別子名に基づいて検出を行うため，識別子名の変更を伴うタイプ 2 のコードクローンの検出は不得手である．

#### F 値

F 値とは，適合率と再現率の総合的な評価として用いられ，適合率と再現率の調和平均によって求められる．

本手法では，Apache HTTPD において F 値が 0.81 であり，関数クローン検出法や CCFinderX より高い F 値が得られた．PostgreSQL においては F 値が 0.69 であり，CCFinderX よりは高い F 値が得られたが，関数クローン検出法より低い F 値となった．また，Python においては F 値が 0.67 であり，関数クローン検出法よりは高い F 値が得られたが，CCFinderX より低い F 値となった．3 つのすべてのプロジェクトの合計の F 値は 0.69 であり，関数クローン検出法と CCFinderX より F 値が高いことが確認できた．

#### 検出時間

検出時間の比較では，表 2.1 の 3 つのプロジェクトに対する検出時間を測定した．本実験の実行環境は，CPU Intel Xeon 2.80GHz 4core，メモリ 16GB，ハードディスクドライブ，OS Windows 10 64bit である．

検出時間の比較結果を表 2.3 に示す．本手法では，検出対象すべてのプロジェクトに対して 3 分以下でコードクローンを検出ができた．また，関数クローン検出法に対して 3∼4 割程度，CCFinderX に対して 4∼8 割程度と，他の手法よりも短時間で検出することが確認できた．

#### ベンチマークに含まれたコードクローンのタイプ

本節では，ベンチマークにおいて保守対象と回答されたコードクローンのタイプ別の個数について説明する．2.4.1 節で実施した，各ツールの検出結果からサンプリングしたコードクローンに対するアンケートにて，保守対象と回答したコードクローンのタイプ（タイプ 1 からタイプ 4）の調査を各回答者に行った．その結果に基づき，各手

表 2.3　検出時間の比較

| 検出対象 | 本手法 | 関数クローン検出法 | CCFinderX |
|---|---|---|---|
| Apache HTTPD | 1m 39s | 4m 7s | 2m 1s |
| PostgreSQL | 2m 27s | 8m 47s | 5m 30s |
| Python | 1m 15s | 3m 33s | 3m 10s |

法が検出した正解クローンのタイプ別の個数と誤検出数を表 2.4 に示す．今回は，保守対象と回答された検出を正解クローン，保守対象と回答されなかった検出を誤検出とする．なお，回答者によって回答が異なる場合，第 1 著者が最終的な判断を行った．

これより，本手法と関数クローン検出法が実際にタイプ 1 から 4 まで検出可能であることが確認できた．また，CCFinderX はタイプ 1 と 2 が検出可能であり，タイプ 3 と 4 は検出できないことが確認できた．表 2.4 は各ツールの検出結果から 90 個ずつ抽出した結果の内訳であり，各タイプの検出数は母集団（各ツールの検出結果）に含まれる絶対数ではないことに注意されたい．例えば，表 2.4 において，CCFinderX のタイプ 1 より本手法のタイプ 1 の数が多いが，CCFinderX と比べて本手法がより多くのタイプ 1 を検出結果に含んでいることを表していない．

### 2.4.2 保守対象と判定されなかったコードクローンの調査

本節では，2.4.1 節で実施したアンケートにて，ツールによって検出されたが保守対象とならないと回答されたコードクローンについて説明する．今回はアンケートにて，3 人中 1 人以下が保守対象となると回答した検出を誤検出とする．本手法，関数クローン検出法，CCFinderX の 3 つの手法にて誤検出した 88 個のコードクローンを調査し，原因を考察した．その結果を表 2.6 に示す．今回は複数回出現した原因を手法ごとに掲載している．

本手法では「2 文字以下の変数の多用」,「if 文の連続」,「同じ識別子の多用」,「出力処理の連続」が原因として確認できた．特に「2 文字以下の変数の多用」と「同じ識別子の多用」は，本手法の識別子名に基づいた検出に起因する．本手法では 2 文字以下の変数を同一のメタワードに置換している．そのため，2 文字以下の変数が多用され

表 2.4　ベンチマークに含まれたコードクローンのタイプ別内訳

| 検出手法 | T1 | T2 | T3 | T4 | 誤検出 | 合計 |
|---|---|---|---|---|---|---|
| 本手法 | 9 | 34 | 27 | 1 | 19 | 90 |
| 関数クローン検出法 | 5 | 29 | 25 | 1 | 30 | 90 |
| CCFinderX | 4 | 47 | 0 | 0 | 39 | 90 |

表 2.5　ベンチマークに含まれたコードクローンの内訳（検出対象ごと）

| | Apache HTTPD | | | | PostgreSQL | | | | Python | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 検出手法 | T1 | T2 | T3 | T4 | T1 | T2 | T3 | T4 | T1 | T2 | T3 | T4 |
| 本手法 | 4 | 14 | 8 | 1 | 4 | 10 | 3 | 0 | 1 | 10 | 16 | 0 |
| 関数クローン検出法 | 1 | 14 | 11 | 0 | 3 | 12 | 10 | 0 | 1 | 3 | 4 | 1 |
| CCFinderX | 2 | 19 | 0 | 0 | 2 | 2 | 0 | 0 | 0 | 26 | 0 | 0 |

ている場合，同一のワードが頻出していると判断し，誤検出の原因となる．また，一般的によく使用される識別子名の場合，処理内容が異なっていても同じ識別子名が用いられる傾向がある点も，誤検出の原因になる．

関数クローン検出法では「ツールの不具合」が原因として多く確認できた．これはC 言語のマクロの処理が適切に行われていない点に起因する．本手法ではマクロの処理を適切に行えているため，ツールの不具合を原因とした誤検出は今回出現しなかった．また，処理の内容は異なるが，二重の繰り返し構文の構造を内部に持つ関数も誤検出として見られた．

CCFinderX では case 節やコードの断片，if 文や代入文の連続の原因による誤検出の例が目立ち，これらは Higo らの指摘と同様の結果が得られた [68]．本手法ではコードブロック単位での検出を行うため，case 節やコードの断片といった保守作業の対象となりにくい誤検出は今回出現しなかった．

また，太字で表記した「if 文の連続」は 3 つの手法で共通して確認できた．

### 2.4.3 コードクローンに対する保守作業の調査

本節では，コードクローンと実際に適用する保守作業の関連性について，本手法，関数クローン検出法，CCFinderX の 3 つの手法との比較を行う．今回は，2.4.1 節で実施したアンケートにて 3 人中 2 人以上が保守対象となると判定したコードクローンに対し，どのような保守作業を適用できるかについて以下のアンケートを行った．

**調査対象**　2.4.1 節で実施したアンケートと同様
**質問内容**　どのような保守作業の対象となるか
**回答方式**　三択（複数回答可）

- 集約
- 同時修正
- その他（自由回答形式）

結果を手法ごとに分けて図 2.5 に示す．なお，回答者によって異なる保守作業が適用された場合，両方の保守作業を適用できることとする．また，その他の保守対象と判

表 2.6　保守対象と判定されなかったコードクローン

| 本手法 | 関数クローン検出法 | CCFinderX |
|---|---|---|
| 2 文字以下の変数の多用 | **if 文の連続** | case 節の断片 |
| **if 文の連続** | 同じ識別子の多用 | **if 文の連続** |
| 出力処理の連続 | 処理内容が異なる繰り返し構文 | コードの断片 |
| 同じ識別子の多用 | ツールの不具合 | 代入文の連続 |

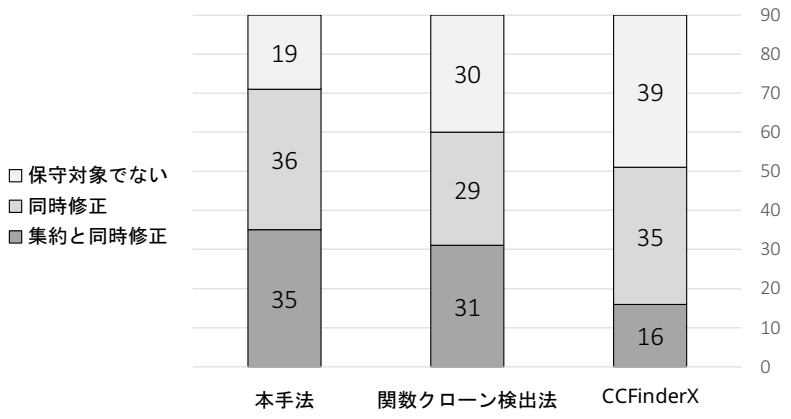


図 2.5　ベンチマークに含まれたコードクローンに対する保守作業

断されたクローンペアについては，回答者と相談し同時修正の対象に含めた．

本手法では 90 個中 71 個のクローンペアが保守対象と判断された．その内，36 個が同時修正の対象，35 個が集約と同時修正の対象となった．関数クローン検出法では 90 個中 60 個のクローンペアが保守対象と判断された．その内，29 個が同時修正の対象，31 個が集約と同時修正の対象となった．CCFinderX では 90 個中 51 個のクローンペアが保守対象と判断された．その内，35 個が同時修正の対象，16 個が集約と同時修正の対象となった．また，いずれの手法においても集約のみの対象と判断されたクローンペアは 0 個であった．

関数クローン検出法と比較して，保守作業の割合は同じであるが，保守対象となる検出数が増加している．関数単位の検出法では集約の対象となりやすい点が長所として挙げられている．本手法では検出粒度をコードブロック単位に小さくしたが，関数単位の検出と保守作業の割合は変わらず，さらに保守作業の対象となりやすい傾向にあることが確認できた．また CCFinderX と比較して，同時修正の対象数は同程度であるが，集約と同時修正両方の対象数は増加している．字句単位の検出法では集約を行うことが困難なクローンが多く検出される点が指摘されていた．本手法では処理内容にまとまりを持ったコードブロック単位の検出を行うため，字句単位の検出法より集約の対象となりやすい傾向にあることが確認できた．以上より字句単位の検出法より集約の対象となりやすく，また関数単位の検出法より保守作業の対象となりやすいと言える．

さらに，本手法においてタイプ別に適用される保守作業の調査を行った．結果を図 2.6 に示す．これより，タイプ 1 は集約と同時修正のどちらの対象にもなりやすく，タイプが上がるにつれて集約の対象にはなりにくいことが分かった．

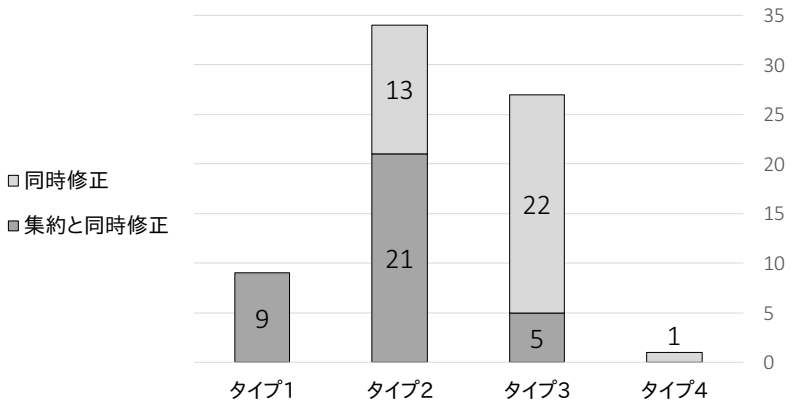


図 2.6　本手法のタイプ別の保守作業

### 2.4.4　検出時間とスケーラビリティの評価

本節では，提案手法の検出規模ごとの検出時間とスケーラビリティの評価について述べる．検出対象の規模の指標にはコメントや空行を除いた LOC を用いることとした．LOC の計測には cloc[*8]を用いた．検出対象は，IJaDataset 2.0[*9]からランダムにファイルを選択し，表 2.7 に示した LOC ごとのサブセットシステムを作成した．IJaDataset とは，2 万以上の Java プロジェクトから成る，365MLOC のビッグデータセットである．本実験の実行環境は，CPU Intel Xeon 2.80GHz 4core，メモリ 32GB，ソリッドステートドライブ，OS Windows 10 64bit である．また，Java 仮想マシンのスタック領域を 1GB，ヒープ領域を 15GB と設定して実行した．

検出時間の評価結果を表 2.7 に示す．これにより，10MLOC 規模の検出対象に対して CCFinderX は検出に約 2 時間かかり，関数クローン検出法ではメモリ不足で

表 2.7　検出規模ごとの検出時間

| LOC | 本手法 | CCFinderX | 関数クローン検出法 |
|---|---|---|---|
| 1K | 1s | 4s | 11s |
| 10K | 2s | 8s | 15s |
| 100K | 16s | 1m 18s | 2m 25s |
| 1M | 3m 1s | 12m 24s | 20m 4s |
| 10M | 29m 9s | 2h 4m 55s | メモリ不足で停止 |

[*8] http://cloc.sourceforge.net
[*9] http://secold.org/projects/seclone

終了したのに対し，本手法では約 30 分で検出可能であることが確認できた．また，1KLOC から 10MLOC の検出規模に応じて線形的に検出時間が増加することも確認できた．

表 2.7 より，本手法は関数クローン検出法や CCFinderX より高速に検出できることが確認できた．また，10MLOC 規模のプロジェクトに対して約 30 分で検出できることも確認できた．特徴ベクトルの次元を $d$，特徴ベクトル集合の大きさを $n$ とした時，クローン検出にかかる時間は $O(dn^2)$ である．しかし，LSH を用いてクラスタリングを行うことで，クローン検出時間は $O(dn^{\rho+1})$ となる．$\rho$ は LSH の鋭敏性を示す値で，LSH に適切なパラメータを与えることで，$\rho \leq 0.2$ に抑えることができる [55]．よって $O(dn^{\rho+1}) < O(dn^2)$ となり，時間計算量は少なくなる．関数クローン検出法と比較して本手法は検出粒度を下げているため $n$ が大きくなるが，LSH を用いることで検出にかかる時間計算量の増加を抑えている．さらに，LSH の実装に用いた FALCONN は高速に LSH を計算できるように実装されたツールであり，クラスタリングを行うツールの変更が検出の高速化の主な要因であると考えられる．また，特徴ベクトルを表現するデータ構造を疎ベクトルとして実装したことにより入出力時間の短縮につながった．入出力時間が検出全体で占める割合も大きいため，特徴ベクトルのデータ構造の変更も検出の高速化に貢献している．

また表 2.7 より，検出規模に応じて線形的に検出時間が増加することも確認できた．これに関しては，本手法や関数クローン検出法ではコードブロック，関数の抽出，CCFinderX では字句の抽出にかかる時間の割合が全ステップ中で多いため，検出規模に応じて線形的に時間が増加すると考えられる．CCFinderX は字句の抽出にかかる時間が長いため，本手法の方が高速に検出することができた．

さらに，関数クローン検出法では 10MLOC の検出中にヒープ領域不足で検出を終了した．ヒープ領域不足は特徴ベクトルの生成中に起こったため，特徴ベクトルを疎ベクトルとして実装していないためメモリの使用効率が悪いためであると考えられる．

さらに，抽象構文木を用いた検出法の Deckard と，我々の知る限り，プログラム依存グラフに基づくクローン検出ツールの中で唯一公開されている Scorpio の 2 つの手法を 10MLOC の検出規模で実験を行った．その結果，7 日間経過しても検出が完了しなかった．そのため現実的な時間内での検出は完了しないとして実験を終了した．このことより，抽象構文木を用いた検出法やプログラム依存グラフを用いた検出法と比較しても，本手法はスケーラビリティの観点から優位性があると言える．

### 2.4.5 ブロッククローンの実例

本節では，2.4.1 節で実施したアンケートにて保守対象のコードクローンと回答されたクローンペアの中から，本手法が検出したブロッククローンの実例を示す．赤色のコード片と緑色のコード片がブロッククローンを表している．

図 2.7 は同じ関数内に存在するタイプ 1 のブロッククローンである．関数クローン検出法では関数単位の検出のため，同じ関数内でコピーアンドペーストを行うことで発生したコードクローンを検出できなかった．

図 2.8 は，図 2.8(a) の 708 行目と 712 行目に文の挿入が行われたタイプ 3 のブロッククローンである．関数単位では，図 2.8(a) の 697 から 699 行目，719 から 723 行目のコード片に対応する部分が図 2.8(b) にない．よって，関数単位では類似していないため関数クローン検出法では検出できなかった．

図 2.9 は，ファイルの出力処理を行うタイプ 4 のブロッククローンである．どちらもファイルの入出力関連の処理を行う関数だが，図 2.9(b) の 364 から 366 行目にて個別の処理を行うため，関数クローン検出法では検出できなかった．

また，字句単位の検出である CCFinderX では，図 2.7 のコードクローンを検出できる．図 2.8 は文の挿入が行われていない部分の検出はできるが，それではコード断片のクローンとなってしまう．よって集約などの保守対象となりづらい．図 2.9 はタイプ 4 のクローンであり，CCFinderX はタイプ 2 のクローンまでしか対応していないため検出することができない．さらに，Deckard では，図 2.7 のコードクローンを検出できるが，図 2.8，図 2.9 は検出することができなかった．なお，唯一公開されているプログラム依存グラフに基づく検出ツールである Scorpio は，C 言語に対応していないため適用できなかった．

ブロッククローンの実例より，関数単位より検出粒度を小さくすることで，関数単位では検出できないが本手法で検出できるクローンを示すことができた．また，コードブロック単位でまとまっているため，集約などの保守対象となりやすいことも実例から確認できた．また，実際に本手法はタイプ 1 から 4 の検出に対応しており，さらに他の手法では検出できなかったコードクローンの検出も可能であった．以上の観点から本手法の有用性が確認できた．

```
248: APU_DECLARE(apr_status_t) apr_rmm_destroy(apr_rmm_t *rmm)
249: {
250:     apr_status_t rv;
251:     rmm_block_t *blk;
252:
253:     if ((rv = APR_ANYLOCK_LOCK(&rmm->lock)) != APR_SUCCESS) {
254:         return rv;
255:     }
256:     /* Blast it all --- no going back :) */
257:     if (rmm->base->firstused) {
258:         apr_rmm_off_t this = rmm->base->firstused;
259:         do {
260:             blk = (rmm_block_t *)((char*)rmm->base + this);
261:             this = blk->next;
262:             blk->next = blk->prev = 0;
263:         } while (this);
264:         rmm->base->firstused = 0;
265:     }
266:     if (rmm->base->firstfree) {
267:         apr_rmm_off_t this = rmm->base->firstfree;
268:         do {
269:             blk = (rmm_block_t *)((char*)rmm->base + this);
270:             this = blk->next;
271:             blk->next = blk->prev = 0;
272:         } while (this);
273:         rmm->base->firstfree = 0;
274:     }
275:     rmm->base->abssize = 0;
276:     rmm->size = 0;
277:
278:     return APR_ANYLOCK_UNLOCK(&rmm->lock);
279: }
```

Apache HTTPD: /srclib/apr-util/misc/apr_rmm.c

図 2.7　同じ関数内に存在するブロッククローン（タイプ 1）

```
690: strop_swapcase(PyObject *self, PyObject *args)
691: {
692:         char *s, *s_new;
693:         Py_ssize_t i, n;
694:         PyObject *newstr;
695:         int changed;
696:
697:         WARN;
698:         if (PyString_AsStringAndSize(args, &s, &n))
699:                         return NULL;
700:         newstr = PyString_FromStringAndSize(NULL, n);
701:         if (newstr == NULL)
702:                         return NULL;
703:         s_new = PyString_AsString(newstr);
704:         changed = 0;
705:         for (i = 0; i < n; i++) {
706:                         int c = Py_CHARMASK(*s++);
707:                         if (islower(c)) {
708:                                         changed = 1;
709:                                         *s_new = toupper(c);
710:                         }
711:                         else if (isupper(c)) {
712:                                         changed = 1;
713:                                         *s_new = tolower(c);
714:                         }
715:                         else
716:                                         *s_new = c;
717:                         s_new++;
718:         }
719:         if (!changed) {
720:                         Py_DECREF(newstr);
721:                         Py_INCREF(args);
722:                         return args;
723:         }
724:         return newstr;
725: }
```

(a) Python: /Modules/stropmodule.c

```
2308: string_swapcase(PyStringObject *self)
2309: {
2310:         char *s = PyString_AS_STRING(self), *s_new;
2311:         Py_ssize_t i, n = PyString_GET_SIZE(self);
2312:         PyObject *newobj;
2313:
2314:         newobj = PyString_FromStringAndSize(NULL, n);
2315:         if (newobj == NULL)
2316:                         return NULL;
2317:         s_new = PyString_AsString(newobj);
2318:         for (i = 0; i < n; i++) {
2319:                         int c = Py_CHARMASK(*s++);
2320:                         if (islower(c)) {
2321:                                         *s_new = toupper(c);
2322:                         }
2323:                         else if (isupper(c)) {
2324:                                         *s_new = tolower(c);
2325:                         }
2326:                         else
2327:                                         *s_new = c;
2328:                         s_new++;
2329:         }

2330:         return newobj;
2331: }
```

(b) Python: /Objects/stringobject.c

図 2.8　文の挿入が行われたブロッククローン（タイプ 3）

```
334: APR_DECLARE(apr_status_t) apr_file_flush(apr_file_t *thefile)
335: {
336:     apr_status_t rv = APR_SUCCESS;
337:
338:     if (thefile->buffered) {
339:         file_lock(thefile);
340:         rv = apr_file_flush_locked(thefile);
341:         file_unlock(thefile);
342:     }
343:     /*
344:      * (コメント省略)
345:      */
346:     return rv;
347: }
```

(a) Apache HTTPD: /srclib/apr/file_io/unix/readwrite.c

```
349: APR_DECLARE(apr_status_t) apr_file_sync(apr_file_t *thefile)
350: {
351:     apr_status_t rv = APR_SUCCESS;
352:
353:     file_lock(thefile);
354:
355:     if (thefile->buffered) {
356:         rv = apr_file_flush_locked(thefile);
357:
358:         if (rv != APR_SUCCESS) {
359:             file_unlock(thefile);
360:             return rv;
361:         }
362:     }
363:
364:     if (fsync(thefile->filedes)) {
365:         rv = apr_get_os_error();
366:     }
367:
368:     file_unlock(thefile);
369:
370:     return rv;
371: }
```

(b) Apache HTTPD: /srclib/apr/file_io/unix/readwrite.c

図 2.9　ファイルの出力処理を行うブロッククローン（タイプ 4）

### 2.4.6 関数クローン検出法と CCFinderX と比較したときの本手法の特徴

評価実験の結果から，本手法が関数クローン検出法や CCFinderX より優れた点は以下の点であると考えられる．

- 保守対象となりにくいブロックの一部をコードクローンとして検出することがなく，また関数よりも小さい単位の検出が可能であるため，字句単位の手法や関数クローン検出手法と比べて，保守（集約や同時修正）対象のコードクローンを多く検出可能である．
- 字句単位の手法や関数クローン検出手法だけでなく，抽象構文木やプログラム依存グラフを用いた手法と比べてもスケーラビリティが高いことから，大規模なソースコード集合に対して適用可能

これらの点から，大規模なソースコード集合に対して，保守対象のコードクローンを検出する場合に本手法が最も適していると考えられる．

一方で，本手法はコードブロック単位で検出を行うため，コードブロックよりも細かい粒度のコードクローンの検出漏れは避けられない．特に，同時修正が必要となる場面では，検出漏れを避けるためにより細かい粒度での検出が適している可能性がある．評価実験の結果では，コードブロック単位での検出のほうが字句単位の検出と比べて，開発者が保守対象として判断しやすいことが示された．これにより，本手法は検出粒度のバランスに優れているといえる．ただし，検出漏れの最小化が最も重要なプロジェクトにおいては，より細かい粒度でクローンを検出する手法や，grep などの文字列検索ツールの利用も検討してもよい．その場合，検出されるクローンの数が増大するため，すべての検出結果の確認を疎かにしないよう注意が必要である．

また，評価実験の結果から，短い変数名が多く出現するなど，変数名に処理内容が反映されてないソースコードに対して本手法は不向きであり，そのようなソースコードに対しては他の手法の利用を検討すべきである．また，時間が十分にあり，かつ 100 万行未満のソースコード集合が対象であれば，本手法に加えて抽象構文木やプログラム依存グラフを用いた手法の利用も合わせて検討すべきである．

### 2.4.7 粗粒度なコードクローン検出法と比較したときの本手法の特徴

同じコードブロック単位の検出手法として，粗粒度なコードクローン検出法 [65] を対象として比較実験を行った．粗粒度なコードクローン検出法とは，タイプ 1 と 2 のブロック単位のコードクローンを検出する手法である．本節ではその実験結果について述べる．

最初に，粗粒度なコードクローン検出法の検出精度を比較した．本実験では，2.4.1

節で作成したベンチマークを用いて適合率，再現率，F 値を求めた．ここでは，ベンチマークにて保守対象と判定されたクローンペアを正解クローン，判定されなかったクローンペアを誤検出と定義する．また，粗粒度なコードクローン検出法が検出したクローンペアを検出クローンと定義する．適合率は，検出クローンの中から，正解クローンと一致した集合と誤検出と一致した集合の和集合の内，正解クローンと一致した集合の割合によって求める．再現率は，正解クローン集合の内，検出クローンの中で正解クローンと一致した集合の割合によって求める．F 値は適合率と再現率の調和平均によって求める．実験の結果，粗粒度なコードクローン検出法の検出精度は表 2.8 のとおりとなった．粗粒度なクローン検出法はタイプ 1 と 2 のコードクローンしか検出しないため，保守対象となりやすく適合率は高い値となった．しかし，タイプ 3 や 4 を検出することができず検出数が少ないため再現率は低い値となり，総合的な評価の F 値は低い値となることが確認できた．

また，粗粒度なコードクローン検出法の検出時間も求めた．実験の結果を表 2.9 に示す．これにより粗粒度なコードクローン検出法は高速に検出ができることが確認できた．粗粒度なコードクローン検出法は，各コードブロックに対して，正規化後（変数名の置換など）の文字列からハッシュ値を求め，同じハッシュ値を持つコードブロックをコードクローンとして検出する．ハッシュ値を用いた比較を行うことで，文字列を用いた比較を行う場合と比べて，小さいコストで 2 つのブロックの比較を行うことができる [65]．

表 2.8　粗粒度なコードクローン検出法の検出精度の評価

| 検出手法 | 検出対象 | 適合率 | 再現率 | F 値 |
|---|---|---|---|---|
| 粗粒度クローン検出法 | Apache HTTPD | **0.91** | 0.14 | 0.24 |
| | PostgreSQL | **0.87** | 0.28 | 0.43 |
| | Python | **0.95** | 0.32 | 0.48 |
| | 合計 | **0.91** | 0.24 | 0.38 |
| 本手法 | Apache HTTPD | 0.90 | **0.74** | **0.81** |
| | PostgreSQL | 0.57 | **0.87** | **0.69** |
| | Python | 0.90 | **0.53** | **0.67** |
| | 合計 | 0.68 | **0.70** | **0.69** |

表 2.9　粗粒度なコードクローン検出法の検出時間

| 検出対象 | 粗粒度クローン検出法 | 本手法 |
|---|---|---|
| Apache HTTPD | 8s | 1m 39s |
| PostgreSQL | 16s | 2m 27s |
| Python | 8s | 1m 15s |

次に，粗粒度なコードクローン検出法が検出したコードクローンのタイプ別の個数について調査した．本実験では，CCFinderX が検出したクローンペア集合に対して，2.4.1 節のアンケートにより得られたコードクローンのタイプ情報を基に，タイプ別の個数を検出対象ごとに算出した．結果を表 2.10 に示す．これにより，粗粒度なコードクローン検出法はタイプ 2 のクローンのみ検出することが分かった．検出法の理論上ではタイプ 1 のコードクローンも検出可能であるが，今回はタイプ 1 のコードクローンは確認できなかった．これは，元々検出対象となっている CCFinderX が検出したコードクローンにタイプ 1 のコードクローンが少なかったことが原因として挙げられる．

さらに，粗粒度なコードクローン検出法の検出規模ごとの検出時間とスケーラビリティについても述べる．本実験では 2.4.4 節と同一のデータセットを用いて実験を行った．実験の結果を表 2.11 に示す．これにより，検出規模に比例して線形的に検出時間が増加することも確認できた．これは，検出時間においてファイル読み込みとコードブロックの抽出に最も時間を占めていることが原因として挙げられる．

最後に，粗粒度なコードクローン検出法の検出したコードクローンに対する保守作業の調査を行った．本実験では，CCFinderX が検出したクローンペア集合に対して，2.4.3 節のアンケートにより得られた保守作業の調査結果をもとに，実際に適用される保守作業の割合と，保守対象と判定されなかった原因を求めた．実際に適用される保守作業は，同時修正の対象となるクローンペアが 15，集約と同時修正の対象となるクローンペアが 3 となった．また，保守対象と判定されなかった原因として，出力処理の連続，代入文の連続，単調な関数呼び出しの 3 例が挙げられた．

表 2.10　粗粒度なコードクローン検出法の検出したコードクローンのタイプ別内訳

| 検出対象 | T1 | T2 | T1, T2 合計 |
|---|---|---|---|
| Apache HTTPD | 0 | 4 | 4 |
| PostgreSQL | 0 | 1 | 1 |
| Python | 0 | 13 | 13 |
| 合計 | 0 | 18 | 18 |

表 2.11　粗粒度なコードクローン検出法の検出規模ごとの検出時間

| LOC | 粗粒度クローン検出法 | 本手法 |
|---|---|---|
| 1K | 0s | 1s |
| 10K | 1s | 2s |
| 100K | 8s | 16s |
| 1M | 1m 15s | 3m 1s |
| 10M | 12m 38s | 29m 9s |

以上の結果から，タイプ 1 や 2 だけでなく，3 と 4 のコードクローンも検出可能で網羅性が高く，検出の正確性と網羅性の総合評価でも高い点が本手法の方が優れていることが確認できた．ただし，粗粒度なコードクローン検出法はコードブロック単位の検出を行うため，トークン単位の検出法である CCFinderX で確認されたようなコード片の断片を検出することはなく，検出の正確性が高くなっている．また，タイプ 1 と 2 のコードクローンのみを検出するため，高速な検出が可能となっている．今回の評価実験から，本手法の補完的な検出方法として粗粒度なコードクローン検出法の利用も検討できる．

## 2.5 考察

### 2.5.1 本手法の拡張性

本手法の実装は，現在 C 言語と Java 言語にのみ対応している．しかし，本手法では ANTLR を用いて構文解析を行っており，ANTLR にて構文解析を行うための文法ファイルが 100 種類以上用意されていることから，他の言語への拡張が容易に可能である．

### 2.5.2 評価実験の妥当性

適合率，再現率，F 値で示される検出精度に関して，本実験では 3 つの C 言語のプロジェクトに対して関数クローン検出法と CCFinderX との比較を行うことによって，本手法の有用性を示した．しかし，今後は他の言語で実装されたプロジェクトに対して適用したり，他のツールとの比較を行ったりするなど，一般性を示す必要性がある．

また，本実験で用いたパラメータの値によって実験結果は変わってくる．特に CCFinderX の実験結果は最小一致トークン数の値が影響している可能性があるが，最小一致トークン数を下げると再現率は上昇するが適合率が低下する．しかし一方で，検出粒度を細かくすると，検出されるコードクローンの数が増え，検出結果の利用が困難になることも報告されている [65]．大規模なソフトウェアに対しては，ある程度まとまったコード片を検出するほうが有用であると著者は考える．

特徴ベクトルの計算方法として本手法では TF-IDF を用いているが，他にも LSI（Latent Semantic Indexing）[44] や，LDA（Latent Dirichlet Allocation）[69] など次元圧縮を行い計算時間の短縮を行った手法がある．またワードの共起頻度に基づいて次元圧縮を行うことで，ワードの類義性も計算可能となる．これらと TF-IDF を比較することで，検出速度と検出精度の観点から比較を行う必要がある．

そして，LSI や LDA を用いて特徴ベクトルを計算する場合，特徴ベクトルの生成や類似度の計算にかかる時間が変化する．たとえば，次元圧縮を行うため特徴ベクトル生成に時間が増加するが，次元削減により類似度計算の時間が減少するなどが考えら

れる．したがって，LSI や LDA などを用いた場合に各ステップにかかる時間を詳細に調査する必要がある．

また本研究では我々が作成したコードクローンベンチマークを対象に検出精度の評価実験を行ったが，他にも大規模なベンチマーク [70] も提案されている．このような他のベンチマークに対しても本手法の有用性が確認できるのか評価する必要がある．さらに今回比較を行っていない手法との比較も行い，本手法の検出結果の特徴についてより詳細に分析する必要もある．

## 2.6 関連研究

構文の類似性に着目した手法として，字句単位の検出手法や，抽象構文木（ソースコードの構文構造を木構造で表したグラフ）を用いた検出手法が存在する．字句単位の検出手法では，ソースコードをトークン列に変換し，共通トークン列をコードクローンとして検出する [58, 71]．また，抽象構文木を用いた検出手法では，ソースコードを抽象構文木に変換し，類似した部分木をコードクローンとして検出する [62, 51]．本研究では，提案手法と字句単位の検出ツールである CCFinderX と比較し，提案手法が適合率や再現率，スケーラビリティにおいて優れていることを確認した．また，スケーラビリティの実験において，抽象構文木を用いた検出ツールである Deckard が 10MLOC のソースコード集合を対象とした実験において現実的な時間で検出処理を終えることができなかったこと，および Deckard が検出できないが提案手法が検出できるコードクローンの実例を確認した．

また，プログラムの処理の類似性に着目した手法として，プログラム依存グラフ（プログラム内の要素間に存在する依存関係を表した有効グラフ）を用いた検出手法が存在する．この手法では，ソースコードからプログラム依存グラフを構築し，類似した部分グラフを探索することによって，タイプ 3 や 4 のコードクローンを検出することが可能である [63, 64, 72]．しかし，プログラム依存グラフの比較の計算コストが高く，検出に時間がかかってしまうという問題点がある．また，プログラム言語毎にプログラム依存グラフを構築する機構を用意する必要性がある．本研究では，プログラム依存グラフを用いた検出ツールである Scorpio [63, 64] が 10MLOC のソースコード集合を対象とした実験において現実的な時間で検出処理を終えることができなかったことを確認した．

LCS (Longest Common Subsequence) アルゴリズムを利用した検出手法として NiCad が存在する [73, 74]．NiCad は，ソースコードの各行を要素とする列に対して LCS アルゴリズムを適用して，固有の行の割合がしきい値以下であればコードクローンとして検出する．しきい値未満であれば，固有の行を許容するため，タイプ 3 のコードクローンを検出できる一方で，タイプ 4 のコードクローン検出については課題が指摘されている [75]．NiCad は文献により実装方法に差異があり，またチューニン

グにより検出速度が大きく異なる [76, 73, 74]．今後，本研究が提案する手法と比較を行う必要があるが，各実装に応じたチューニングを適切に行いながら，比較実験を行う必要があると考えられる．

山中らのツール [46] 以外の関数クローン検出ツールとして MeCC [77] がある．MeCC は，記号実行を行うことによって，ソースコード中の各関数が終了した時点における抽象的なメモリの状態の予測を行う．そして，メモリの状態が類似した関数をコードクローンとして検出する．山中らの論文 [46] において，山中らの関数クローン検出ツールの方が MeCC より正確に関数クローンを検出できることが確認されている．本論文の実験では，山中らの関数クローン検出ツールと比較して，提案手法の方が適合率や再現率，スケーラビリティにおいて優れている点が多いことを示した．

Marcus らは，クエリとして与えられたソースファイルと類似した部分を，LSI（Latent Semantic Indexing）[44] を用いてソースコード全体から検索する手法を提案している [78]．彼らが提案する手法はクエリを与える必要があるため，本研究の提案手法とは目的が異なる．しかし，提案手法においても LSI を利用し識別子間の潜在的意味を解析することで，再現性を向上させることができる可能性がある．

我々の研究グループでは，トークン列が等価なファイルを高速に検出するツール FCFinder の開発を行った [79]．本研究では，識別子や予約語の類似性に着目し，ブロック単位のコードクローンを検出する手法を提案した．本研究が提案する手法は，2 つのブロックに含まれるトークン列が異なっていても，識別子や予約語の集合が類似していれば，それらブロックはブロッククローンとして検出される．

Duala-Ekoko らは，クローン領域を抽象的に記述し，バージョン間でのクローンの移動を追跡し管理する手法を提案している [80]．ソフトウェアの開発過程においてコードクローンの変更管理を行う場合，ソースコードの編集により行情報は変更されるため，従来の行情報を用いたクローン領域の記述方法ではコードクローンを追跡できない．そこで彼らは，クローン領域を抽象的に記述する手法を提案した．彼らの提案手法はコードクローン追跡のための手法であり，コードクローン検出を目的とする本研究とは目的が異なる．彼らの手法では，構文，構造，語彙の情報を組み合わせてクローン領域を抽象的に記述することでクローン追跡を可能にしている．しかし，本手法はクローン追跡の必要性がないためファイルと行の情報で記述されたクローン領域をコードブロックとして扱っている．

## 2.7 まとめと今後の課題

本研究では，TF-IDF と LSH を利用したブロッククローン検出手法の提案を行った．本手法では，構文解析を行いコードブロックの抽出を行い，コードブロック中の識別子や予約語に利用されている単語からワードを抽出する．そして，TF-IDF を利用して各ワードに対する重みを計算し，その重みを特徴量として各コードブロックを

特徴ベクトルに変換する．その後，特徴ベクトル間の類似度を計算することによって，意味的に処理が類似したブロッククローンの検出を行う．また，multi-probe LSH を cross-polytope LSH に組み合わせた手法を用いてあらかじめ特徴ベクトルのクラスタリングを行うことによって，高速なブロッククローンの検出を実現した．

評価実験では，3 つの C プロジェクトに対し，検出精度と検出時間の観点から，関数クローン検出法と CCFinderX の 2 つの手法と比較を行った．比較した結果，本手法が高い精度かつ高速にコードクローンを検出することが確認できた．また，スケーラビリティの評価では，1KLOC から 10MLOC の検出規模に応じて，線形的に検出時間が増加することを確認できた．

今後の課題として，2.5.2 節にて述べたように以下が挙げられる．

- 今回は特徴ベクトルの計算に TF-IDF を用いたが，LSI（Latent Semantic Indexing）[44] や，LDA（Latent Dirichlet Allocation）[69] といった次元圧縮を行う手法がある．これら次元圧縮を用いた手法との比較を行う必要がある．
- LSI や LDA などを用いた場合，検出の各ステップにかかる時間を詳細に調査する必要がある．
- 他の大規模プロジェクトに対して適用し，本手法の有用性を評価する必要がある．さらに，他の検出手法との比較を行う必要がある．

# 第 3 章

# 情報検索技術と深層学習を用いたコード片類似性判定法の比較調査

## 3.1 まえがき

コード片の類似性判定法はソフトウェア工学における重要な基礎技術である．類似したコード片を見つける作業はソフトウェア開発や保守において重要であり，コードクローン検出 [43] やコード片検索 [81, 82] などではコード片の類似性判定法が使用される．コードクローンとはソースコード中に含まれる互いに一致または類似した部分を持つコード片である．一般的に，コードクローンの存在はソフトウェアの保守を困難にすると言われている．膨大な量のコードクローンを目視で見つけることは困難であり，コードクローンを自動的に検出する手法が提案されている [8, 42]．またコード片検索は，再利用可能なコード片を特定するために使用する [83]．既存のコード片を再利用することで，ソフトウェア開発の生産性および信頼性の向上が期待できる．さらにコード片検索を用いて，再利用元のライセンス記述を調べたり，より優れた脆弱性対策がされているコード片を探したりでき，再利用の安全性を高めることができる．

コード片の類似性判定法では，構文的な類似性に基づく判定法と，意味的な類似性に基づく判定法の 2 種類がある [84]．既存研究の多くはコード片の構文的な類似性に基づいて判定する [58, 85, 66]．一方で，意味的な類似性に基づいた判定法は少なく，様々な課題が残されている．コード片の意味的な類似性判定法として Zhao らは DeepSim という手法を提案した [84]．DeepSim は制御フローグラフとデータフローグラフの情報をもとに，深層学習モデルを用いてコード片を判定する．DeepSim は類似したコード片を高い精度で判定できる一方で，実行速度が遅いという課題もある．

また我々は既存研究で，情報検索技術に基づくコードクローン検出法を提案した [86]．この手法は情報検索技術の一種である TF-IDF（Term Frequency-Inverse Document Frequency）[44] を用いてコード片をベクトル化し，ベクトル空間上で距離が近いコード片をコードクローンとして検出する．ベクトル空間上の距離尺度はコサ

イン類似度を用いる．コードクローン検出にベクトル表現を用いることで，構文や字句単位で類似していなくてもベクトル空間で距離が近ければコードクローンとして検出できる．この手法は高速に検出できる一方で，構文的な類似性が低いコードクローンの検出漏れが多い課題がある [87, 88]．

構文的な類似性が低くても意味的に類似しているコード片を，高い精度で高速に判定できるコード片の類似性判定が望ましい．コード片の類似性判定法では，情報検出技術や深層学習が広く使われている．藤原らは深層学習を用いたソースコード検索において情報検索技術の一種である BoW（Bag of Words）と Doc2Vec[89] の精度の比較を行っているが，他の情報検索技術について評価していない [90]．また，再帰型ニューラルネットワークを用いてコード片をベクトル化してコード片の類似性を判定する手法はあるが [91, 92]，本研究の目的である情報検索技術を用いたコード片のベクトル化は網羅的に調査されていない．これらの研究では本研究と目的が異なるため，判定精度が高く実行速度が速い，情報検出技術と深層学習の組み合わせは明らかになっていない．そこで本研究では，意味的なコード片の類似性判定において，判定精度と実行速度の観点から有効な情報検索技術と深層学習の組み合わせを調査する．本調査では，情報検索技術に基づきコード片をベクトル表現に変換し，変換したベクトル表現を深層学習モデルに入力することで，コード片が類似しているか否かを判定する．これにより，情報検索技術と深層学習を組み合わせたコード片の類似性判定法を実現する．また本調査で使用する情報検索技術と深層学習の組み合わせを，深層学習を用いた既存手法と比較する．

本調査では，データセットとして Google Code Jam（以降 GCJ）[*1] と BigCloneBench（以降 BCB）[70] を用いて調査する．GCJ は Google が開催している競技プログラミングコンテストであり，同じ問題に正解したソースコードは類似コードとみなす．BCB はコードクローンの大規模ベンチマークであり，600 万以上のクローンペア（処理内容が類似しているコードクローンの対）と 26 万以上の非クローンペアが登録されている．これらのデータセットについて調査した結果，情報検索技術の一種である LSI（Latent Semantic Indexing）[44] と深層学習モデルを組み合わせた手法が，適合率，再現率，F 値においてより高い値となった．また，この組み合わせは実行速度が最も速いことも確認した．

以降，3.2 章では，本研究の背景について述べる．3.3 章では，本研究の調査手法について述べる．3.4 章では，本研究の調査結果について述べる．3.5 章では，調査結果から得られた考察について述べる．3.6 章では，関連研究について述べる．最後に，3.7 章でまとめと今後の課題について述べる．

[*1] https://codingcompetitions.withgoogle.com/codejam/

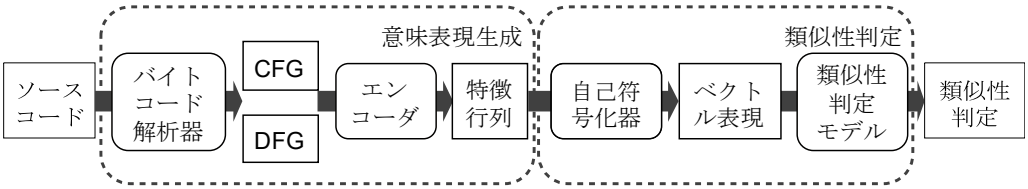


図 3.1　DeepSim の概要

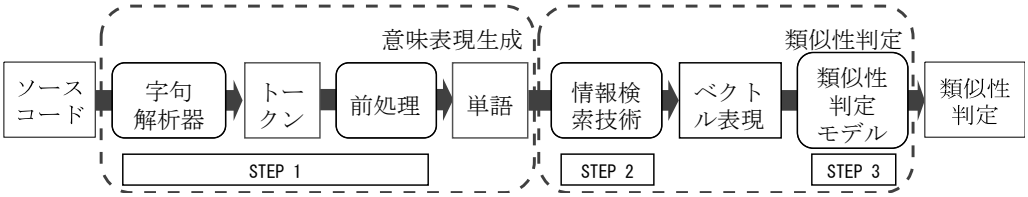


図 3.2　調査に用いるコード片類似性判定法の概要

## 3.2　背景

### 3.2.1　情報検索技術に基づくコードクローン検出法

コード片の類似性判定法が用いられる場面の 1 つに，コードクローン検出がある [8]．コードクローンとは，ソースコード中に含まれる互いに一致または類似した部分を持つコード片である．

コードクローンを自動的に検出する手法は，これまでにも数多く提案されている [8, 42]．我々は先行研究において，情報検索技術に基づくコードクローン検出法 CCVolti[*2]を提案した [86]．CCVolti はまず，情報検索技術の一種の TF-IDF を用いてコード片をベクトル化する．そして，与えられた 2 つのベクトル表現間のコサイン類似度を求める．これによりコード片の処理内容が意味的に類似しているか判定し，1.5.1 項で説明したタイプ 4 までのコードクローンを検出する．この手法は高速に検出できる一方で，構文的な類似性の低いコードクローンの検出漏れが多い課題が指摘されている [87, 88]．

### 3.2.2　深層学習を用いたコード片類似性判定法

深層学習を用いたコード片の類似性判定法として，Zhao らは DeepSim[84] を提案した．DeepSim の概要を図 3.1 に示す．この図が示すように，DeepSim は前半の意味表現生成過程と後半の類似性判定過程に分かれる．意味表現生成過程では，制御フローグラフ（以降 CFG）とデータフローグラフ（以降 DFG）を解析し，特徴行列に

[*2] https://github.com/k-yokoi/CCVolti

変換することでコード片の意味表現を生成する．また，類似性判定過程では，ニューラルネットワークの一種である自己符号化器 [93] を用いて，特徴行列の情報を最もよく表すベクトル表現に変換する．その後，ベクトル表現を類似性判定モデルに入力し，2 つのコード片が類似しているか否かの二値分類を行う．DeepSim は高い精度でコード片の類似性を判定する．一方で DeepSim はモデルの学習に時間がかかるという課題がある [84].

## 3.3 調査手法

本研究では，コード片類似性判定法に用いる情報検索技術と深層学習の組み合わせについて調査する．また，情報検索技術と深層学習の組み合わせについて，深層学習を用いた既存手法である DeepSim[84] と FA-AST[94] の 2 つの結果と比較する．本章ではその調査手法について述べる．

### 3.3.1 調査目的とリサーチクエスチョン

3.2.1 節で述べた情報検索技術に基づくコードクローン検出法 [86] は高速な検出が可能だが，構文的な類似性が低いコードクローンの検出漏れが多い課題がある [87, 88]. 一方で，3.2.2 節で述べた深層学習を用いた類似性判定法は高い精度でコード片の類似性を判定するが，モデルの学習に時間がかかる課題がある [84]．コード片の類似性判定はソフトウェア開発や保守において広く使われており，判定精度が高く実行速度が速い手法が望ましい．しかし，判定精度と実行速度の観点でどの情報検出技術と深層学習の組み合わせが，判定精度が高く実行速度が速い手法かが明らかになっていない．そこで，本調査では 2 つのリサーチクエスチョンを設定した.

RQ1　高い精度で類似性を判定する情報検索技術と深層学習の組み合わせは何か？
RQ2　短時間で類似性を判定する情報検索技術と深層学習の組み合わせは何か？

この 2 つのリサーチクエスチョンを解くことで，精度と実行速度の 2 つの観点から有用な情報検索技術と深層学習の組み合わせを明らかにする.

### 3.3.2 調査に用いるコード片類似性判定法

本調査で用いるコード片類似性判定法は以下の 3 つのステップで実行する．手法の概要を図 3.2 に示す.

STEP 1　ソースコード解析を用いた前処理を行い，コード片を単語列に変換する
STEP 2　情報検索技術に基づき単語列をベクトル表現に変換する
STEP 3　ベクトル表現を類似性判定モデルに入力し，深層学習によりコード片の類似

性を判定する

**STEP 1：ソースコード解析を用いた前処理**

最初に，入力コード片に対して字句解析を行いトークン列に変換する．字句解析には構文解析器生成系 ANTLR[*3]が生成した字句解析器を用いる．

次にトークン列に対して前処理を行い単語列に変換する．本調査では以下の前処理を行う．

- 予約語と識別子以外を除去
- 識別子名をキャメルケースやスネークケースを基に分割
- 分割後の識別子をすべて小文字に正規化

これらの前処理は既存研究と同じ方法を用いる [86]．識別子分割や正規化は，情報検索技術によりソースコードを解析する際に有用な手法として用いられる [95]．

**STEP 2：情報検索技術に基づくベクトル化**

次に STEP1 で生成した単語列に対して，情報検索技術に基づくベクトル化によりベクトル表現に変換する．情報検索技術に基づくベクトル化には Python ライブラリ gensim を用いる [96]．

本研究では情報検索技術に基づくベクトル表現として，LSI（Latent Semantic Indexing），LDA（Latent Dirichlet Allocation），Doc2Vec，WV-avg（Word2Vec average）の 4 つのベクトル表現を調査対象として選択した．WV-avg は単語ベクトル Word2Vec[97, 98] の平均ベクトルを意味する．調査対象のベクトル表現は，先行研究 [87] において採用されたベクトル表現を参考に選択した．また，先行研究で使用されているベクトルの表現のうち，BoW（Bag of Words）と TF-IDF はベクトルの次元数が変動するため，FT-avg は WV-avg より再現率が低く計算速度も遅い [87] ため，本調査では使用しない．ただし Doc2Vec の実装として PV-DBoW（Distributed Bag of Words version of Paragraph Vector）と PV-DM（Distributed Memory version of Paragraph Vector）の 2 つの異なるアルゴリズムが提案されている [89] ため，本調査ではこの 2 つのアルゴリズムを別個のベクトル表現として使用する．したがって，LSI，LDA，PV-DBoW，PV-DM，WV-avg の 5 つのベクトル表現を調査対象とする．なお Word2Vec の実装としても複数のアルゴリズムが提案されているが，本研究では SGNS（skip-gram algorithm with negative sampling）[98] を用いる．

本研究で対象とするベクトル表現と，そのベクトル表現に与える主なハイパーパラメータを付録 1 に示す．ハイパーパラメータは Python ライブラリ gensim のデフォルト値を参考に決定した．ハイパーパラメータのチューニングにより判定精度が上昇

[*3] https://www.antlr.org/

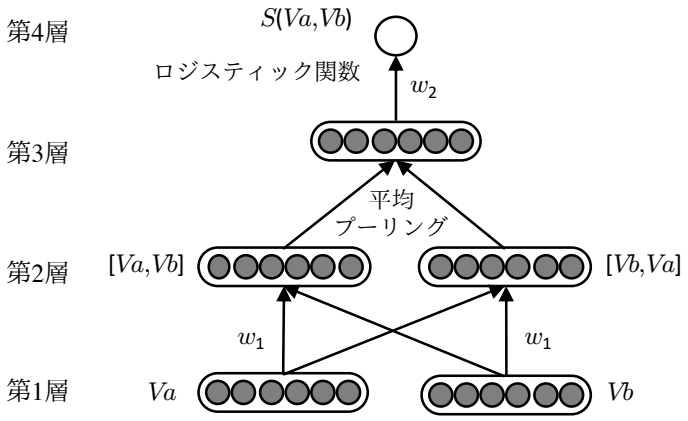


図 3.3　コード片類似性判定モデルのアーキテクチャ

する可能性がある一方で，実行速度が遅くなる懸念もある．また，ハイパーパラメータチューニングは多くの計算リソースと時間を必要とし，チューニングプロセスに膨大な時間がかかる．ソフトウェア開発に適用する際，ハイパーパラメータチューニングにかかる計算リソースおよび時間と得られる精度向上のバランスを考慮し，実用性の観点からチューニングを行わないと判断される場合がある．そこで本調査では，ハイパーパラメータチューニングを行わずにライブラリのデフォルト値における判定精度と実行速度を調査する方針とした．

**STEP 3：深層学習を用いた類似性判定モデル**

最後に，STEP2 で生成したベクトル表現に対して，深層学習モデルを用いてコード片の類似性を判定する．本研究では順伝播型ニューラルネットワークを基にコード片類似性判定モデルを設計した．コード片類似性判定モデルのアーキテクチャを図 3.3 に示す．図 3.3 が示すように，2 つのコード片から得られたベクトル表現 $V_a$ と $V_b$ を第 1 層に入力する．第 2 層では，入力した 2 つのベクトルを $[V_a, V_b]$，$[V_b, V_a]$ と異なる順序で連結する．このとき 2 つの連結ベクトルは重み $w_1$ を共有して計算される．第 3 層は平均プーリング層である．2 つの層からなる第 2 層の出力値の平均値を第 3 層への入力として計算する．2 つの異なる順序でベクトルを連結し平均プーリングを行うことは，2 つのベクトルの対称性を成立させる役割を果たしている．最後は，出力層としてロジスティック関数を用いてコード片の類似性を出力する．ロジスティック関数は入力を $u$ として次の関数であらわされる（$f(u) = \frac{1}{1+e^{-u}}$ ）．出力値は 0 から 1 の間であり，出力値が 0.5 以上であれば類似している，0.5 未満であれば類似していないと判定する．活性化関数は ReLU，損失関数は CrossEntropy，最適化アルゴリズムは Adam を用いる．このニューラルネットワークアーキテクチャを実現するために，

本研究では TensorFlow[*4] と Keras[*5] を用いて実装した．

### 3.3.3 比較調査対象

本調査では，情報検索技術と深層学習の組み合わせの 5 種類と 3.2.2 節で説明した DeepSim の，合計 6 種類を比較調査対象とする．

#### 情報検索技術と深層学習の組み合わせ

3.3.1 節で説明した 2 つの RQ に答えるため，3.3.2 節で説明した 5 種類のベクトル表現と 3.3.2 節で説明した深層学習モデルの組み合わせを比較調査対象とする．本論文では，深層学習モデルに入力した 5 種のベクトル表現を区別するために以下の表記を用いる．NN はニューラルネット（Neural Network）の略である．

- LSI+NN (Latent Semantic Indexing + Neural Network)
- LDA+NN (Latent Dirichlet Allocation + Neural Network)
- PV-DBoW+NN (Distributed Bag of Words version of Paragraph Vector + Neural Network)
- PV-DM+NN (Distributed Memory of Words version of Paragraph Vector + Neural Network)
- WV-avg+NN (Word2Vec average vector + Neural Network)

深層学習モデルで用いたハイパーパラメータは付録 2 に示す．ハイパーパラメータは Python ライブラリ gensim のデフォルト値を参考に決定した．

#### 深層学習を用いた既存手法

情報検索技術と深層学習の組み合わせを，深層学習を用いた既存手法とも比較した．本調査では Zhao らの DeepSim[84] と Wang らの FA-AST+GMN[94] の 2 つの既存手法を比較調査対象とする．DeepSim は CFG と DFG を解析して得た意味表現から自己符号化器を用いてベクトル表現を生成し，深層学習モデルを用いてコード片の類似性を判定する．また FA-AST+GMN は抽象構文木に CFG と DFG の情報を加えて拡張したグラフ表現からグラフニューラルネットワークを用いてベクトル表現を生成し，コサイン類似度を用いてコード片の類似性を判定する．

DeepSim と FA-AST+GMN は 3.3.4 節で説明する Google Code Jam と BigCloneBench[70] をデータセットして評価を行っている．そのため，深層学習を用いたコード片類似性判定法の中からこの 2 つの既存手法を採用した．なお，Wang らは論文中 FA-AST+GGNN と FA-AST+GMN の 2 つの手法を評価しているが，

[*4] https://www.tensorflow.org/
[*5] https://keras.io/

FA-AST+GMN の方が F 値が高かった [94] ため，本調査では FA-AST+GMN のみを比較調査対象とする．DeepSim と FA-AST+GMN で用いたハイパーパラメータは付録 2 に示す．

### 3.3.4 対象データセット

本調査では，オンラインジャッジサイトと広く使われているベンチマークの 2 種類のデータセットを用いて調査する．オンラインジャッジサイトは意味的な類似性判定のベンチマークとして信頼性がある [84]．また広く使われているベンチマークと比較することは，他の手法と比較しやすいことから重要である．これら 2 種類のデータセットを用いることで，より網羅的な調査が可能となる．さらに本調査においてはデータセットが広く公開されているという観点から，オンラインジャッジサイトとして Google Code Jam（以降 GCJ）を，ベンチマークとして BigCloneBench（以降 BCB）[70] を用いて調査した．

GCJ は Google[*6]が実施している競技プログラミングコンテストであり，提出されたソースコードは Google によって挙動を検証されている．これにより，異なるプログラマが提出したソースコードも同一の挙動をすることを保証されており，これらは意味的に類似したソースコードとみなせる．本研究では 12 種の問題から集めた 1,669 個の提出コードを用いて実験した [84]．同一問題に対して提出されたソースコードは類似コード，異なる問題に対して提出されたソースコードは非類似コードとみなすことができる．なお，実際に提出されたソースコードによって関数分割の粒度に差がある．例えば 1 つの関数にまとめて処理を記述したソースコードもあれば，複数の関数に分割して記述したソースコードもある．そのため本研究では提出されたソースコードを main 関数にインライン展開し，展開後の main 関数を実験の対象とした．

BCB は大規模なコードクローンベンチマークである [70]．BCB では約 6 万のコード片に対して，クローンペア（処理内容が類似しているコードクローンの対）または非クローンペアとしてタグ付けされおり，600 万以上のクローンペアと約 26 万の非クローンペアが登録されている．ただし BCB は更新されており，この研究で使用した最新データセットのクローンペア総数は，BCB の原著で報告された数値と若干異なる．本研究では DeepSim の評価実験と条件を揃えるため，クローンペアまたは非クローンペアとタグ付けされていないコード片と，5 行未満のコード片をデータセットから除去した．除去した理由は，行数の小さいコード片はコードクローンとして保守対象になりにくく，コードクローンの評価実験において対象外にされることが多いからである [99]．この除去により，クローンペアまたは非クローンペアにタグ付けされたコード片は，約 5 万に減少した．除去後のデータセットには，約 594 万のクローンペアと約 18 万の非クローンペアが含まれている．BCB ではコードクローンを 5 つの

[*6] https://about.google/

タイプに分類する [70]．BCB で用いる分類では，Roy らが提案したコードクローンの 4 つタイプ分類 [43] のうち，タイプ 3 とタイプ 4 のコードクローンを定量的に分類する．BCB に登録されているクローンペアのほとんどは構文的類似性が低い WT3/T4 である．コードクローン各タイプの個数と割合を表 3.1 に示す．

### 3.3.5 リサーチクエスチョンの調査方法

**調査 1：精度の調査**

RQ1 に回答するため，本調査では GCJ と BCB の両方のデータセットに対しての精度を調査した．本研究では再現率，適合率，F 値を精度評価指標とする．再現率とは正解集合に対して実際に正しく正解と推定された割合を指し，網羅性を示す指標である．また適合率とは正解として推定された集合に対して真に正しい割合を指し，正確性を示す指標である．一般的に再現率と適合率はトレードオフの関係にあり，一方の値が高くなるともう一方の値が低くなる．そこで，再現率と適合率の総合的な評価指標として F 値が用いられる．F 値は再現率と適合率の調和平均により求められる．

さらに情報検索技術と深層学習の組み合わせで意味的な類似性の判定精度が向上することを確認するため，BCB のデータセットに対してタイプ別の調査を行った．精度は F 値を用いて比較する．なお情報検索技術のみを用いた手法では，3.3.2 項で説明した 5 種類のベクトル表現に対して，コサイン類似度 0.9 の閾値以上になるコード片を類似していると判定した．閾値を 0.9 とした理由は既存研究においてコードクローンの検出精度の観点から優れた値だったからである [46]．

本研究では 10 分割交差検証を用いて調査対象の類似性判定モデルの精度を推定する．10 分割交差検証ではデータセットを 10 分割し，1 つをテスト用データ，残りの 9 つを学習用データとして検証する．この手順をテスト用データと学習用データの組み合わせを変えて 10 回繰り返す．こうして得られた結果を平均し，類似性判定モデルの精度を推定する．ただし FA-AST+GMN は 10 分割交差検証を行っておらず，学習データ: 検証データ: テストデータの比を 8:1:1 の割合でデータセットを分割して評価している．またクローンペアと非クローンペアの数が 1:1 になるよう，クローンペアを減らす調整も行っている [94]．FA-AST+GMN の評価方法が本調査と異なるため，参考記録として表に掲載する．

表 3.1　コードクローンの各タイプの個数と割合（BCB）

| タイプ | T1 | T2 | ST3 | MT3 | WT3/T4 |
|---|---|---|---|---|---|
| 個数（個） | 17,398 | 3,734 | 12,032 | 53,616 | 5,860,619 |
| 割合（%） | 0.3 | 0.1 | 0.2 | 0.9 | 98.5 |

**調査 2：実行時間の調査**

RQ2 に回答するため，本調査では GCJ のデータセットを用いて実行時間を調査した[*7]．実行時間として，類似性判定モデルの学習にかかる時間と，類似性判定モデルによる類似性の推定にかかる時間の 2 つを調査する．本調査では GCJ の 9 割を学習し，1 割を推定するために所要した時間を測定した[*8]．

さらに意味表現生成過程の実行時間を測定した[*9]．図 3.1 が示すように，DeepSim は意味表現生成過程においてバイトコード解析を用いてソースコードから CFG と DFG を生成し，その後エンコーダを用いて特徴行列を生成する．一方で図 3.2 のように調査で用いるコード片類似性判定法の意味表現生成過程では，字句解析によるトークン分割後に前処理によって単語列を生成する．生成方法の違いによる実行時間の差を明らかにするため，表 3.2 に示す 6 つのオープンソースソフトウェアプロジェクトを用いて意味表現生成に要する時間を測定した．6 つのプロジェクトは MvnRepository[*10]に含まれており，ソースコードとバイトコードの両方が公開されている．なお調査で用いるコード片類似性判定法はソースコードを解析対象とし，DeepSim はコンパイル後のバイトコードを解析対象とした．意味表現生成過程の実行時間の測定は LSI+NN の 5 回平均より求めた．

表 3.2　意味表現生成の時間評価に用いた対象プロジェクト

| プロジェクト | バージョン | ファイル数 | LOC |
|---|---|---|---|
| ANTLR | 4.7.2 | 233 | 96,043 |
| Apache Ant | 1.10.5 | 787 | 92,219 |
| Apache Commons Lang | 3.8.1 | 153 | 27,646 |
| Apache Log4j | 1.2.17 | 213 | 21,050 |
| JUnit | 4.12 | 195 | 9,317 |
| Guava | 27.0.1 | 573 | 86,542 |

[*7] BCB は関数単位のコードクローンのベンチマークとして作られており，ソースコードのコンパイルができない．DeepSim のバイトコード解析器ではソースコードのコンパイルが必要なため，BCB を用いた実行時間調査は比較していない．

[*8] 実験環境は Intel Xeon E5 2.7GHz 4 コア CPU，NVIDIA Quadro 5000 GPU，32GB メモリのワークステーション

[*9] 実験環境は Intel Xeon E5 2.7GHz 4 コア CPU，32GB メモリのワークステーション

[*10] https://mvnrepository.com/

## 3.4 調査結果

### 3.4.1 精度の調査結果

GCJ を用いた精度評価の結果を表 3.3 の左側に示す[*11]．表 3.3 が示すように，LSI+NN の再現率 93%，適合率 96%，F 値 0.94 と情報検索技術と深層学習の組み合わせの中で最も高い精度を示した．LDA+NN は再現率 54%，適合率 61%，F 値 0.55 と最も低い精度となった．また DeepSim の再現率 81%，適合率 71%，F 値 0.76 と比較して，LSI+NN，PV-DBoW+NN，WV-avg+NN の 3 つの組み合わせが再現率，適合率，F 値ともに DeepSim の結果を上回った．PV-DM+NN の再現率は DeepSim を上回ったが，適合率と F 値は DeepSim より低い値となった．LDA+NN は再現率，適合率，F 値ともに DeepSim より低い値となった．一方で FA-AST+GMN は再現率 97%，適合率 99%，F 値 0.98 と，LSI+NN よりも高い精度となった．

BCB を用いた精度評価の結果を表 3.3 の右側に示す[*11]．なお DeepSim の値は有効桁数 2 桁までしか掲載されていないため，本調査で調べた組み合わせとは表中の有効桁数が異なる．再現率に関しては 5 つの組み合わせのすべてが 99.9% と高い値となった．適合率に関しては 5 つの組み合わせの中では PV-DBoW が 99.9% と最も高く，他の 4 つの手法も 99.5% 以上の値が得られた．また DeepSim は再現率 97%，適合率 98%，F 値 0.98，FA-AST+GMN は再現率 94%，適合率 96%，F 値 0.95 となった．これらと比較して，5 つの組み合わせのすべてが再現率，適合率，F 値いずれも

表 3.3　GCJ と BCB を用いた精度評価

| | GCJ | | | BCB | | |
|---|---|---|---|---|---|---|
| 手法 | 再現率 | 適合率 | F 値 | 再現率 | 適合率 | F 値 |
| LSI+NN | **0.93** | **0.96** | **0.94** | **0.999** | 0.995 | 0.997 |
| LDA+NN | 0.54 | 0.61 | 0.55 | **0.999** | 0.995 | 0.997 |
| PV-DBoW+NN | 0.88 | 0.86 | 0.86 | **0.999** | **0.999** | **0.999** |
| PV-DM+NN | 0.89 | 0.68 | 0.75 | **0.999** | 0.998 | 0.998 |
| WV-avg+NN | 0.91 | 0.90 | 0.88 | **0.999** | 0.998 | 0.998 |
| DeepSim | 0.82 | 0.71 | 0.76 | 0.98 | 0.97 | 0.98 |
| FA-AST+GMN | 0.97 | 0.99 | 0.98 | 0.94 | 0.96 | 0.95 |

（注）：FA-AST+GMN の論文における精度評価の条件設定 [94] が本調査と異なるため，参考記録として掲載する．

[*11] DeepSim と FA-AST+GMN の再現率，適合率，F 値は各論文より引用した．本調査と DeepSim は同じ条件設定 [7] で調査しているが，FA-AST+GMN は条件設定が異なる [21] ため参考記録として掲載する．

DeepSim と FA-AST+GMN の結果を上回った．

さらに BCB におけるクローンタイプごとの F 値の調査結果を表 3.4 に示す．この表より，特に WT3/T4 において，情報検索技術と深層学習の組み合わせた手法の F 値が向上していることが確認できた．これにより，情報検索技術と深層学習の組み合わせることで，構文的類似性が低いクローンも高い精度で判定できることが分かった．

RQ1 への回答

情報検索技術と深層学習を用いたコード片類似性判定法の中で LSI+NN の精度が最も高い

### 3.4.2 実行時間の調査結果

GCJ を用いた実行時間の評価結果を表 3.5 に示す．また情報検索技術に基づくベクトル化に要する時間を明らかにするため，学習時間の内訳も表 3.5 に示す．表 3.5 より，学習時間に関して LSI+NN が 210 秒と最も高速だった．一方で PV-DM が 533 秒と最も遅く，LSI+NN と比較して約 2.6 倍の時間がかかっている．学習時間の内訳では，ベクトル化に所要する時間は LSI+NN が 0.6 秒と最も高速であり，類似性判定に所要する時間は LDA+NN が 201 秒と最も高速であった．一方でベクトル化に所要する時間は PV-DM が 318 秒と最も遅く，類似性判定に所要する時間は PV-DBoW+NN と WV-avg+NN が 216 秒と最も遅かった．推定時間に関

表 3.4　BCB におけるクローンタイプごとの F 値

| クローンタイプ | T1 | T2 | ST3 | MT3 | WT3/T4 |
|---|---|---|---|---|---|
| LSI+NN | 1.00 | 1.00 | 0.999 | 0.997 | 0.996 |
| LDA+NN | 1.00 | 1.00 | 0.999 | 0.997 | 0.997 |
| PV-DBoW+NN | 1.00 | 1.00 | 0.999 | 0.997 | 0.996 |
| PV-DM+NN | 0.999 | 0.999 | 0.997 | 0.996 | 0.992 |
| WV-avg+NN | 1.00 | 1.00 | 0.999 | 0.997 | 0.998 |
| LSI | 0.999 | 0.988 | 0.833 | 0.216 | 0.003 |
| LDA | 0.999 | 0.986 | 0.889 | 0.624 | 0.108 |
| PV-DBoW | 0.999 | 0.987 | 0.702 | 0.056 | 0.000 |
| PV-DM | 0.685 | 0.832 | 0.310 | 0.021 | 0.000 |
| WV-avg | 0.999 | 0.994 | 0.938 | 0.762 | 0.144 |
| DeepSim | 0.81 | 0.71 | 0.76 | 0.97 | 0.98 |
| FA-AST+GMN | 1.00 | 1.00 | 0.998 | 0.982 | 0.946 |

（注）：FA-AST+GMN の論文における精度評価の条件設定 [94] が本調査と異なるため，参考記録として掲載する．

しては LSI+NN と LDA+NN が 21 秒と最も高速だった．一方で PV-DBoW+NN, PV-DM+NN，WV-avg+NN の 3 つの組み合わせが 25 秒と最も遅く，LSI+NN や LDA+NN と比較して約 1.2 倍の時間がかかっている．

また DeepSim の学習時間に関しては 70,503 秒と，約 20 時間かかった[*12]．最も高速な LSI+NN と比較して約 336 倍の時間がかかっている．なお DeepSim はベクトル化を行わないため，学習時間の内訳の記載は無い．また DeepSim の推定時間 27 秒と比較し，5 つの組み合わせのすべてが短時間で推定が完了した．

図 3.1，図 3.2 の左枠で囲われた意味表現生成過程の実行時間の調査結果を表 3.6 に示す．この表より，LSI+NN は DeepSim の 27˜44% の実行時間で意味表現を生成できた．さらに DeepSim による ANTLR の解析時には，Java 仮想マシンのヒープ領域不足により意味表現の生成を完了できなかった．なお，Java 仮想マシンの最大ヒープサイズを初期値の 8GB から 16GB まで増やして再度実行したが結果は変わらなかった．

表 3.5　GCJ を用いた実行時間評価（秒）

| 手法 | 学習時間 | 推定時間 | 学習時間内訳 | |
|---|---|---|---|---|
| | | | ベクトル化 | 類似性判定 |
| LSI+NN | **210 秒** | **21 秒** | **0.6 秒** | 209 秒 |
| LDA+NN | 228 秒 | **21 秒** | 27 秒 | **201 秒** |
| PV-DBoW+NN | 224 秒 | 25 秒 | 8 秒 | 216 秒 |
| PV-DM+NN | 533 秒 | 25 秒 | 318 秒 | 215 秒 |
| WV-avg+NN | 256 秒 | 25 秒 | 40 秒 | 216 秒 |
| DeepSim | 70503 秒 | 27 秒 | - | - |

表 3.6　意味表現生成過程の実行時間（秒）

| プロジェクト | DeepSim | LSI+NN |
|---|---|---|
| ANTLR | メモリ不足 | 13 秒 |
| Apache Ant | 42 秒 | 12 秒 |
| Apache Commons Lang | 20 秒 | 6 秒 |
| Apache Log4j | 15 秒 | 4 秒 |
| JUnit | 9 秒 | 4 秒 |
| Guava | 29 秒 | 12 秒 |

[*12] DeepSim の論文に掲載されている学習時間の 13525 秒 [84] と比較し，本調査では約 5.2 倍の学習時間を要している．実験環境の性能差により学習時間に差が生じた．

RQ2 への回答

情報検索技術と深層学習を用いたコード片類似性判定法の中で LSI+NN の実行時間が最も短い

## 3.5 考察

### 3.5.1 調査結果 1: 精度の比較

情報検索技術に基づくベクトル表現を用いた 5 種類の類似性判定法の中では，3.4.1 節の表 3.3 より LSI+NN が再現率, 適合率, F 値ともに最も高く, 次いで WV-avg+NN が高かった．その反面，LDA+NN は再現率 54%，適合率 61% と両者とも最も低い値となった．LDA はベイズ統計に基づく確率的トピック生成モデルによってコード片のベクトル表現を求めるが，プログラミング言語は自然言語と比べてトピック数が少ないため，コード片のベクトル表現を求める上で LDA は効果的でない可能性がある．また，PV-DBoW+NN，PV-DM+NN，WV-avg+NN が LSI+NN より再現率と適合率ともに低い値となった．これらの 3 つの類似性判定法は，機械学習を用いてベクトル表現を生成する．しかし，3.3.2 節で述べたとおり，精度と実行速度のトレードオフの観点から本調査では Python ライブラリのデフォルト値を参考にしてベクトル表現を生成する際のハイパーパラメータを設定した．本調査ではハイパーパラメータを調整しなかったことが，上記の 3 つの類似性判定法が LSI+NN より低い精度となった原因の可能性がある．ただし，精度と実行速度はトレードオフの関係にあること，またハイパーパラメータのチューニングにかかる時間と精度向上の効果のバランスを考慮すると，デフォルト値での精度評価はベクトル表現の素の性質を評価する上で有用である．

BCB を用いた精度評価では，5 種類全ての組み合わせにおいて再現率，適合率，F 値ともに 0.99 以上と大きな差はなかった．さらに BCB におけるクローンタイプごとの判定精度の比較も行った．情報検索技術と深層学習を組み合わせた手法は，構文的類似性が低いクローンも高い精度で判定できることを示した．情報検索技術を用いた手法は検出漏れが多い課題があったが，この調査結果より情報検索技術と深層学習を組み合わせることの有用性が明らかになった．

GCJ と BCB の精度結果を比較すると，BCB の方が精度が高い傾向にある．これはデータセットを構成するクローンの特徴が，精度の差に繋がっている．BCB に含まれる WT3/T4 クローンは，一致する行数が 50% 以下であるものの，同様の構造をしているクローンが多く含まれている [84]．一方で，GCJ はさまざまなプログラマが開発したソースコードが提出されており，さまざまな構造のコード片の類似性を求める必要がある．また，提出の速度が求められる競技プログラミングにおいて，ソースコードの理解のしやすさは二の次にされることが多く，同一問題に提出されたソース

コードもプログラマによって異なる識別子名をつけられる可能性がある．したがってGCJ の方が類似性を判定することが難しく，BCB の方が GCJ に比べて F 値が高くなる傾向にある．

### 3.5.2 調査結果 2：実行時間の比較

情報検索技術に基づくベクトル表現を用いた 5 種類の類似判定法の中では，3.4.2 節の表 3.5 より学習時間にばらつきがあるのに対し，推定時間の差は少なかった．さらに学習時間の内訳をみると，ベクトル化に所要する時間が学習時間の差に寄与していることが分かった．

ベクトル化に所要する時間に関しては，LSI+NN が 0.6 秒と非常に短時間であった．2 番目の PV-DBoW+NN も 8 秒であり，残りの手法は 20 秒以上であることから，LSI+NN がベクトル化の観点から非常に高速であることがわかる．これらにより，LSI+NN が最も有用性の高い手法であることが確認できた．また，付録 2 が示すように PV-DBoW+NN，PV-DM+NN，WV-avg+NN のベクトル生成のためのエポック数は全て 20 に固定し，ベクトル化にかかる時間を公平に比較した．しかしベクトル化のアルゴリズムによって，ベクトル化に所要する時間は大きく異なる結果となった．

LSI+NN が高速である理由として，LSI は主成分分析を用いてベクトルの次元圧縮を行うことが挙げられる．特に，主成分分析は行列計算に置き換えることで高速に計算可能なアルゴリズムが提案されており [100]，高速にベクトル化を行える．

### 3.5.3 情報検索技術と深層学習の組み合わせの比較

GCJ を用いた精度評価では LSI+NN が再現率，適合率，F 値ともに最も精度が高かった．また BCB を用いた精度評価では大きな差はないため，精度の観点から LSI+NN がコード片の類似性判定法に適していると言える．さらに学習時間と推定時間のいずれにおいても LSI+NN が最も高速であり，実行時間の観点からも LSI+NN がコード片の類似性判定法に適している．

以上の精度と実行速度の 2 つの観点から，LSI+NN が最もコード片の類似性判定法に適していることが判明した．したがって情報検索技術の一種である LSI と深層学習の組み合わせが，コード片類似性判定法として最も効果的な組み合わせである．

### 3.5.4 情報検索技術と深層学習を用いたコード片類似性判定法と既存手法の比較

DeepSim と比較したところ，適合率，再現率，F 値の精度の観点と，学習や推定に所要する実行速度の観点から，LSI+NN の有用性が明らかになった．また FA-

AST+GMN と比較したところ，GCJ における精度評価に関して LSI+NN より高い精度となった．これは，Wang らの論文と本調査の評価方法が異なることが理由として挙げられる．Wang らは 10 分割交差検証を行っておらず，さらにクローンペアの数を減らして非クローンペアと数を揃えるなどの調整を行っており，本調査と条件設定が異なるため参考記録として掲載している．

調査で用いたコード片類似性判定法が高い精度を得られた理由として，人間はコードを書く際にソースコードに処理の意味を持たせることが挙げられる．特にソフトウェアの保守性を高めるために，他人が読んでも理解しやすいソースコードを記述することが求められ，その処理内容を理解できるようにソースコードを記述する．本研究の実験により，人間がソースコードに持たせた処理内容の意味を情報検索技術に基づくベクトル表現が充分に表現できることが分かった．

DeepSim は CFG と DFG を解析した特徴行列から自己符号化器を用いてコード片のベクトル表現を求める．一方で調査で用いたコード片類似性判定法はコード片中の出現単語から，情報検索技術を用いてベクトル表現を求める．つまりベクトル表現を求めるまでの過程の違いにより，調査で用いたコード片類似性判定法は高速な学習が可能になった．特に情報検索技術の一種である LSI にて用いられる主成分分析は，行列計算に置き換えて高速に計算するアルゴリズムが提案されており，ネットワークの学習のために反復計算が必要な自己符号化器と比較して高速に学習ができる．

### 3.5.5 コード片類似性判定の実例

ここでは，BCB 上で正しく類似性判定できたクローンおよび正しく判定できなかったクローンの具体例を示し，その差異を定性的に述べる．

最初に，本調査で用いた 5 つの類似性判定法で正しくクローンと判定した例を示す．図 3.4 と図 3.5 のコード片は，ファイルをコピーする処理を行うコードクローンである．“File”，“Input”，“Output”，“Stream”，“src”，“dest”など共通してあらわれる文字が多く，情報検索技術を用いた手法でクローンと判定しやすい．

次に偽陽性の例を示す．図 3.4 は単にファイルをコピーする処理に対し，図 3.6 は圧縮ファイルを展開する処理を行う．これらは BCB 上で非クローンペアとして登録されているが，本調査で用いた 5 つの類似性判定法はクローンと判定した．“File”，“Input”，“Output”，“Stream”，“src”などが共通して出現するため，類似していると誤って判定された可能性がある．

最後に偽陰性の例を示す．図 3.7 と図 3.8 は Web ページを取得する処理を行うクローンとして BCB に登録されている．しかし，本調査で用いた 5 つの類似性判定法はクローンでないと判定した．これらには共通してあらわれる文字が少ないため，情報検索技術を用いた手法では類似性を判定できない可能性がある．

以上のように，本調査で用いた類似性判定法は情報検索技術を用いているため，構

```
public static void copyFile3(File srcFile, File destFile)
    throws IOException {

  InputStream in = new FileInputStream(srcFile);
  OutputStream out = new FileOutputStream(destFile);

  byte[] buf = new byte[1024];
  int len;
  while ((len = in .read(buf)) > 0) {
    out.write(buf, 0, len);
  } in .close();
  out.close();
}
```

図 3.4　コード片 1: ファイルをコピーする処理（1）

```
static void copy(String src, String dest)
    throws IOException {

  File ifp = new File(src);
  File ofp = new File(dest);
  if (ifp.exists() == false) {
    throw new IOException(
      "file '" + src + "' does not exist");
  }
  FileInputStream fis = new FileInputStream(ifp);
  FileOutputStream fos = new FileOutputStream(ofp);
  byte[] b = new byte[1024];
  int readBytes;
  while ((readBytes = fis.read(b)) > 0)
    fos.write(b, 0, readBytes);
  fis.close();
  fos.close();
}
```

図 3.5　コード片 2：ファイルをコピーする処理（2）

文的に類似性が低いコードクローンの類似性を判定できる．一方で，共通した文字列に影響を受けて誤ってコードクローンと判定したり，共通した文字列が少ないことにより誤ってコードクローンでないと判定したりする欠点がある．

```
public void prepare() throws IOException {

  this.extractDirectory = TempFileFactory.get()
    .createTempDirectory("pdash-compressed-wd", ".tmp");
  File srcZip = getTargetZipFile();
  InputStream in = new FileInputStream(srcZip);
  if (isPdbk(srcZip))
    in = new XorInputStream( in , PDBK_XOR_BITS);
  boolean sawEntry = false;
  try {
    ZipInputStream zipIn =
      new ZipInputStream(new BufferedInputStream( in ));
    ZipEntry e;
    while ((e = zipIn.getNextEntry()) != null) {
      sawEntry = true;
      if (!e.isDirectory()) {
        String filename = e.getName();
        File f = new File(extractDirectory, filename);
        if (filename.indexOf('/') != -1)
          f.getParentFile().mkdirs();
        FileUtils.copyFile(zipIn, f);
        f.setLastModified(e.getTime());
      }
    }
  } finally {
    FileUtils.safelyClose( in );
  }
  if (!sawEntry) throw new ZipException("Not a valid ZIP
file: " + srcZip);
}
```

図 3.6　コード片 3：圧縮ファイルを展開する処理

```
public static String downloadWebpage3(String address)
    throws ClientProtocolException, IOException {

  HttpClient client = HttpClientBuilder.create().build();
  HttpGet request = new HttpGet(address);
  HttpResponse response = client.execute(request);
  BufferedReader br = new BufferedReader(
    new InputStreamReader(
      response.getEntity().getContent()));
  String line;
  String page = "";
  while ((line = br.readLine()) != null) {
    page += line + "¥n";
  }
  br.close();
  return page;
}
```

図 3.7　コード片 4：WEB ページを取得する処理（1）

```
public boolean open() {

  try {
    URL url = new URL(resource);
    conn = url.openConnection();
    in = new BufferedReader(
      new InputStreamReader(conn.getInputStream()));
  } catch (MalformedURLException e) {
    System.out.println(
      "Uable to connect URL:" + resource);
    return false;
  } catch (IOException e) {
    System.out.println(
      "IOExeption when connecting to URL" + resource);
    return false;
  }
  return true;
}
```

図 3.8　コード片 5：WEB ページを取得する処理（2）

### 3.5.6 妥当性の脅威

本研究では交差検証によりモデルが過学習していないことを確認しているが，学習とテストを同じデータセット内にて行っている．学習したデータセット以外でも高い精度が得られる，より汎化性能が高い類似性判定モデルの作成は課題である．本研究では，学習とテストで異なるデータセットを使用した精度調査も行った．BCB で学習し GCJ でテストした結果は，再現率 0.95，適合率 0.21，F 値 0.34 となった．また GCJ で学習し BCB でテストした結果は，再現率 0.63，適合率 0.97，F 値 0.76 となった．これらは表 3.3 の結果より F 値が低い．つまり学習したデータセット以外では精度が低くなることが分かる．これは，データセット内のコード片には偏りが存在することが理由として挙げられる．実際に BCB は 10 種類の機能のコード片しか含まれておらず [70]，また GCJ のデータセットには 12 種類の問題しか含まれていない．しかしすべての機能を持つコード片のデータセットの作成は困難である．学習したデータセット以外でも高い精度が得られる，より汎化性能が高いモデルの作成は課題である．

またハイパーパラメータに関する懸念もある．3.4.1 節にて PV-DBoW+NN と PV-DM+NN の適合率は 86%，68% と，PV-DM の方が適合率が低い．ハイパーパラメータのエポック数を PV-DM は PV-DBoW より増やす場合が多いが，本調査では 20 に揃えた．そのためコード片の意味を充分に学習できず，PV-DM の適合率が低下した可能性がある．上記以外の手法においても，ハイパーパラメータを調整することにより精度が向上する可能性がある．

さらには，本研究で調査したベクトル表現と深層学習モデル以外に，より有用性の高い組み合わせがある可能性がある．本研究では，gensim ライブラリに採用されているベクトル表現から調査対象を選択した．本研究の調査対象外のベクトル表現も調査対象に加えることで，さらに有用性の高い組み合わせを見つけられる可能性がある．

本調査では FA-AST+GMN の精度評価結果を論文より引用したが，調査実験の条件設定が異なる [21] ため参考記録として掲載した．FA-AST+GMN らは 10 分割交差検証を行っておらず，さらにクローンペアの数を減らして非クローンペアと数を揃えるなどの調整を行っており，本調査と条件設定が異なる．今後，本調査と条件設定を揃えて FA-AST+GMN の精度を評価することで，また異なる結果が得られる可能性がある．

本調査では，5 行未満のコード片をデータセットから除去した．除去した理由は，Choi らの報告によると，行数の小さいコード片はコードクローンとして保守対象になりにくく，コードクローンの評価実験において対象外にされることが多いからである [99]．しかし，Choi らの報告では，保守作業としてリファクタリングを想定しており，保守作業として同時修正を行う場合は，5 行未満のコード片も対象となる可能性がある．また，コード片の検出粒度を細かくすることで，再現率が向上し，保守対象とし

ての有用性が高まる可能性がある．しかし一方で，検出粒度を細かくすると，検出されるコードクローンの数が増え，検出結果の利用が困難になることも報告されている [65]．大規模なソフトウェアに対しては，まとまった行のクローンを検出するほうが有用であると著者は考える．

## 3.6 関連研究

深層学習を用いたコード片類似性判定法として White らの手法と Wei らの手法がある．White らは RtvNN というコード片類似性判定法を提案した [91]．RtvNN はトークン列と抽象構文木の情報から再帰型ニューラルネットワーク [101] を用いてベクトル表現を求め，ユークリッド距離を用いてベクトル表現の類似性を判定する．また，Wei らは CDLH というコード片類似性判定法を提案した [15]．CDLH は字句と構文の情報から抽象構文木ベースの LSTM（Long Short-Term Memory）を用いてハッシュコードを求め，高速に類似性を学習する．コード片の類似性判定にはハッシュコードのハミング距離を用いる．RtvNN と CDLH は，深層学習を用いてベクトル表現を求めるが，類似性判定には深層学習モデルを使用しない．さらに BCB を対象に上記の 2 つの手法と DeepSim の精度評価を比較した結果，DeepSim より再現率と適合率ともに低かった [84]．

我々は情報検索技術に基づく様々なベクトル表現をソースコードに適用し，コードクローン検出における各ベクトル表現の特徴を調査した [87, 88]．これらのベクトル表現を用いて BCB[70] に対してコードクローン検出を行い，その再現率を比較することで，よりコード片の意味を表すベクトル表現を調査した．また，コード片の類似性を判定する距離尺度として，コサイン類似度と Word Mover's Distance[102] の 2 つの距離尺度を比較した．一方で本研究では，コード片の類似性を判定するために深層学習モデルを用いている．本研究は，情報検索技術と深層学習の効果的な組み合わせを調査した点で先行研究と異なる．

我々の研究グループでは深層学習を用いたソースコード分類手法を比較調査した [103]．入力された 1 つのソースコードを機能に応じて分類するタスクを設定し，そのタスクを高い精度で解く深層学習モデルを明らかにした．一方，本研究では入力された 2 つのソースコードが類似しているか否かの判定を深層学習モデルのタスクとして設定した．深層学習モデルが解くタスクの観点で，先行研究と本研究は異なる．

また同じく我々の研究グループでは，深層学習モデルとして回帰モデルを用いたコードクローン検出法を提案した [104]．深層学習を用いた既存のコードクローン検出法に対し，深層学習モデルを回帰モデルに変更し汎化性能が高くなることを明らかにした．深層学習モデルへの入力は，既存のコードクローン検出法に則りソースコードメトリクスを用いている．一方，本研究では情報検索技術に基づくベクトル表現を深層学習モデルの入力として用いる．深層学習モデルへの入力の観点で，先行研究と本

研究は異なる．

## 3.7 まとめと今後の課題

本研究では，情報検索技術に基づくベクトル表現と深層学習の効果的な組み合わせについて，判定精度と実行速度の観点から調査した．調査実験では，5 種類のベクトル表現間での比較と，深層学習を用いた既存手法との比較を行った．調査の結果，情報検索技術に基づくベクトル表現として LSI を用いた手法が，最も高精度かつ高速であることが分かった．また既存手法の DeepSim と比較して，LSI を用いた手法が再現率，適合率，F 値ともに高く，学習時間も約 350 倍高速であることが確認できた．既存手法の FA-AST+GMN と比較して，GCJ を用いた精度評価において LSI を用いた手法より FA-AST+GMN の方が再現率，適合率，F 値ともに高かった．ただし調査実験の条件設定が異なるため FA-AST+GMN の精度は参考記録として扱う．

今後の課題として，以下の 3 点が挙げられる．

- 学習したデータセット以外でも高い精度が得られる，より汎化性能が高いモデルの作成
- 本研究で用いたベクトル表現以外の調査
- FA-AST+GMN の精度を，本研究の調査と条件設定を合わせて評価

# 第 4 章

# 段階的再構築における依存関係分析を用いた費用対効果の試算

## 4.1 まえがき

モダナイゼーションとは，長年にわたって稼働してきたレガシーシステムを最新の技術やアーキテクチャを用いて刷新することを指す．ビジネス上の変化への迅速な対応や保守開発工数の削減のため，レガシーシステムのモダナイゼーションに対する企業の需要は大きい．モダナイゼーションにはいくつかの課題があるが，その 1 つが失敗のリスクの高さである．実際に，多くの時間と費用を費やしても，モダナイゼーションが完了しなかったり，失敗したりする事例が報告されている [47]．特に，企業でモダナイゼーションを実施するシステムはビジネス上重要である場合が多いため，失敗リスクを最小限に抑える必要がある．この失敗リスクを軽減する戦略として，システムの一部を切り出す段階的再構築が提案されている [47, 48]．段階的再構築では，旧システムを新システムに段階的に置き換える．これにより，移行に失敗した場合，最初からやり直す必要はなく，失敗した時点からやり直せばよいため，移行失敗のリスクを抑えながらモダナイゼーションを行える．

モダナイゼーションのもうひとつの課題として，費用対効果の試算の難しさがある [18]．企業の経営陣は短期的な投資収益率を重視する傾向があり，一度モダナイゼーションの投資が決定されると，その投資回収が優先される [18]．また，人間は将来の利益よりも目先の損失を避ける傾向があることを知られており，ソフトウェア開発に関する意思決定でも同様であることが示されている [105]．そのため，企業においては要件定義の前工程で行われるシステム化計画 [49] において，モダナイゼーションの費用対効果を見積もり，ユーザー企業とベンダー企業の間で合意を形成することが重要である [50]．特に，システム化計画工程において行われる見積りは，試算とよばれる [50]．近年では，レガシーシステムのモダナイゼーションの研究では，技術的な側面だけでなくビジネスの側面を考慮することにも重点が置かれている [106, 107]．

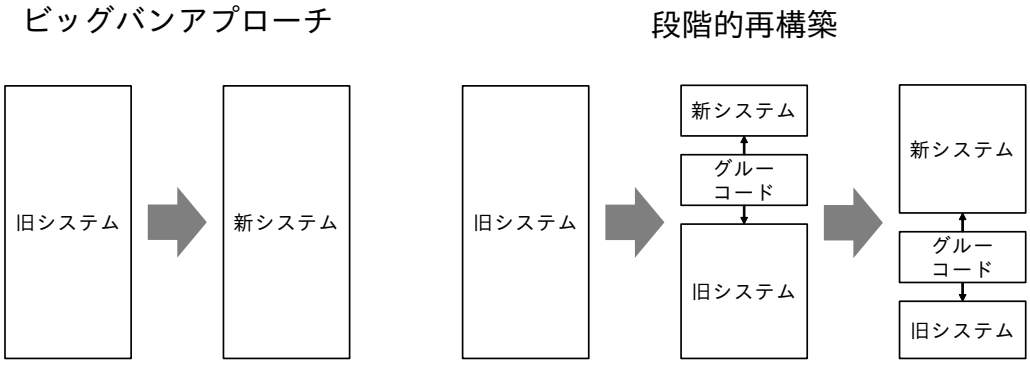


図 4.1　ビッグバンアプローチと段階的再構築

モダナイゼーションの失敗リスクを軽減する段階的再構築においても，費用対効果の試算は難しい．我々の知る限り，段階的再構築を実施するための費用や開発工数，またその後の効果を統計的に集計したデータは公開されていないため，段階的再構築における費用対効果を見積もることは困難である．実際，著者が所属する企業で段階的再構築を実施した事例では，計画当初の見積りと実績値に乖離が生じている．またリファクタリング [40] の費用対効果を見積もるための研究は複数存在するが [108, 109, 110, 111, 112]，我々の知る限り，段階的再構築における費用対効果を見積もるための研究は行われていない．

そこで本研究では，システムの依存関係分析を用いて段階的再構築の費用対効果を試算する手法を調査する．具体的には，過去に段階的再構築を実施した大規模な金融システムを対象にこの手法を適用し，試算の妥当性を検証する．本研究では，システムの規模と依存関係の情報を活用し，費用として段階的再構築に必要な工数を，効果として削減可能な保守開発工数を試算し，実績値との乖離を比較する．また，対象システムの関係者にヒアリングを行い，適用手法の妥当性を定性評価する．本研究の主な貢献は，以下のとおりである．

- 大規模な産業システムに対して，段階的再構築の費用対効果を試算した．
- 費用対効果の実績値と試算値を比較し，結果を考察した．
- 対象システムの関係者にヒアリングを行い，費用対効果試算手法の妥当性を考察した．

以降，4.2 章では，本研究の背景について述べる．4.3 章では，本研究で適用対象とした段階的再構築を実施したプロジェクトおよびシステムについて述べる．4.4 章では，3 章で述べたプロジェクトに適用した費用対効果試算手法について述べる．4.5 章では，試算結果と実績値の調査について述べる．4.6 章では，関連研究について述べる．最後に，4.7 章でまとめと今後の課題について述べる．

## 4.2 背景

### 4.2.1 モダナイゼーション

モダナイゼーションとは，数十年にわたって稼働してきたレガシーシステムを最新の技術やアーキテクチャを用いて刷新することを指す．ビッグバンアプローチ [113] や段階的再構築 [48] など，長年にわたっていくつかのモダナイゼーション手法が提案されてきた．図 4.1 はビッグバンアプローチと段階的再構築の概要を示す．図 4.1 の左側が示すように，ビッグバンアプローチはシステム全体を一度に新システムへ移行するため，移行失敗のリスクが高い．ビッグバンアプローチの移行失敗のリスクを回避するため，段階的再構築という戦略が提案されている [47, 48]．図 4.1 の右側に示されているように，段階的再構築の中心的な考え方は，旧システムと新システムから複合システムを構築することである．段階的再構築では，旧システムを新システムに段階的に置き換えるため，移行失敗のリスクを軽減する．しかし，その一方で，複合システムとしての複雑性に伴い，以下の課題が増大する．

1. 旧システムと新システム間の通信制御が複雑になる．
2. 旧システムと新システムのデータ整合性の維持が困難になる．

旧システムから新システムを切り出す場合，新システムと旧システム間の依存関係が分断される．この分断により，切り出し前は関数呼び出しでつながっていた処理が，切り出し後は遠隔手続き呼び出しを行うこととなる．しかし，旧システムで採用されている技術の多くは新システムとは大きく異なる可能性がある．そのため，新旧のシステム間で遠隔手続き呼び出しをする際に，古い通信プロトコルのサポートが必要となり，通信制御が複雑になる．また，新旧システムの間でデータの整合性を維持できない場合，システムの破損リスクが生じる．

これらの課題を解決するために，新旧システム間の通信を変換するためのグルーコードを配置することで，新旧システムを切り離す必要がある．グルーコードとは，異なるソフトウェアコンポーネントやシステム間での連携を可能にするコード片やスクリプトを指す．これにより，異なるプログラミング言語や OS，ハードウェアを持つ旧システムと新システム間での通信制御やデータ整合性が確保される．この設計パターンを腐敗防止層 [114] ともいう．

グルーコードを介して旧システムと新システムを連携した後は，旧システムから新システムに置き換えられる割合を徐々に増やしていくことで，新システムへの段階的な移行が実現する．このように段階的再構築は移行失敗のリスクを軽減する一方で，通信方式を変換するためにグルーコードを導入する必要があり，グルーコードを追加で開発する工数が発生する．

### 4.2.2 費用見積り

見積り作業では，規模，工数，工期，品質（信頼性，性能など），費用などのさまざまな要素を見積もる [115].

見積りは，類推法，積み上げ法，パラメトリック法の 3 つに大きく分類される．類推法とは，過去の類似プロジェクトの実績を基礎に見積もる方法である．類推法は簡便で計画初期の見積りに適している一方，過去の実績プロジェクトの背景や制約，特徴などが明らかでないと適用が困難である．独立行政法人情報処理推進機構が公開している開発規模や工数などの実績値 [116] を使用し，類推法を適用できる．しかし，情報処理推進機構が集計した開発プロジェクトには，新規開発や再開発などの種別が示されている一方で，段階的再構築を実施したかどうかの情報が含まれてないため，段階的再構築の見積りに使用することは難しい.

積み上げ法は，プロジェクトの成果物の構成要素を洗い出し，それぞれに必要な工数などを見積もって積み上げる方法である．積み上げ法は見積りの精度が高い一方，プロジェクトの構成要素としての作業項目を WBS（Work Breakdown Structure），成果物の構成要素としてのサブシステムやコンポーネントなどを事前に洗い出しておく必要がある．したがって，不確実なものが多い計画段階での見積りでは使用することが難しい.

パラメトリック法は，工数などを目的変数として，説明変数に規模や要因などを設定し，数学的な関数として定式化する方法である．パラメトリック法では，工数と規模が正比例（工数 $= \alpha \times$ 規模）するものや，工数と規模の累乗が比例（工数 $= \alpha \times$ 規模$^{\beta}$）するものが主に利用されている．工数と規模の累乗が比例する関係式は，COCOMO 法 [117] でも採用されている．情報処理推進機構は，工数と規模が正比例するパラメトリック法に基づいてソフトウェア開発の定量データを収集，分析したデータを公開している [116]．本研究では，パラメトリック法の考え方に基づいて工数を見積もる．

### 4.2.3 見積りの時期と誤差

見積り時期によっては，見積り結果と最終的な実績値との間で誤差が発生する．特に，早期の段階の見積りになればなるほどプロジェクト全体に不確実な点が多くなるため，誤差が大きくなる傾向にある．しかし，ユーザー企業ではシステム開発の予算確保や計画実行の可否を判断するために，早期の段階での見積りが必要である．その結果，不確実性が大きい段階での見積りが最後までベンダー企業の束縛となり，プロジェクト成功の阻害要因になっているという課題がある．そこで，不確実性のある段階での見積りで確定せずに，工程ごとに多段階での見積りを実施し，段階的に見積りの精度を上げていくことが推奨されている [50]．図 4.2 は見積りと時期の誤差を示す．この図が示すように，システム化の方向性検討では仮試算，システム化計画では試算，

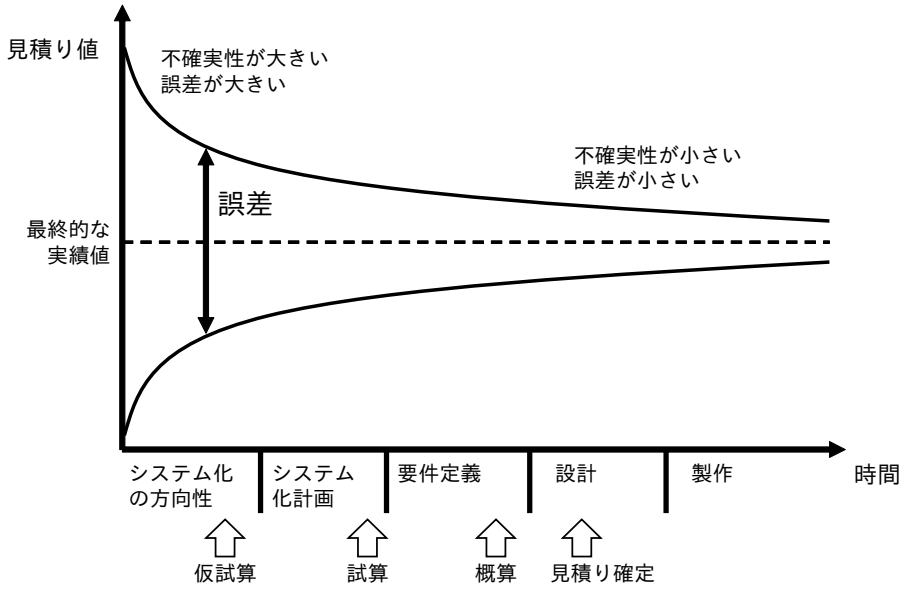


図 4.2　見積りの時期と誤差

**（注）** 文献 [50, 118] の図に基づき作成

要件定義では概算と呼ばれる見積りを行い，設計を終えた段階で確定見積りとする [50].

一般的に，多段階での見積りにより見積り結果と実績値の誤差が発生するリスクは抑えられると言われているが，著者は，早期の見積りと実績値の乖離について下記のような事例を経験している．

- 要件定義工程に入ってから，費用対効果の見積りが不十分であることが発覚し，システム化の方向性検討まで後戻りした事例
- 費用の見積りと実績値の乖離が大きいことを過度に危惧し，システム化の方向性検討から先に進めない事例
- モダナイゼーションの効果を適切に見積もれないことから，システム化方向性検討から先に進めない事例

このような事例から，システム化の方向性検討やシステム化計画において費用対効果の見積り精度の向上が重要であることを示している．そこで本研究では，システム化計画の工程で使用することを想定した試算手法について調査する．

## 4.3　適用対象

本章では，依存関係分析を用いた費用対効果の試算手法を適用した対象プロジェクトについて説明する．

### 4.3.1 対象プロジェクトの説明

対象システムは，著者が現在勤める企業が保守開発を担当していた金融機関の基幹システムである．対象プロジェクトの概要を表 4.1 に示す．段階的再構築を実施時点での対象プロジェクトの規模は，約 4.6MLOC であった．

対象システムの保守性向上を目的として，対象プロジェクトが計画された．対象プロジェクトでは，最終的に対象システムの一部機能を約 870KLOC の Java プログラムとして切り出す段階的再構築が実施された．段階的再構築のための実際の開発工数は，確定見積りによる開発工数を 27.5% 超過した．超過した原因を社内で分析した結果，設計段階で，4.2.1 節で述べた段階的再構築による複雑性の課題を充分に考慮されていなかったことが大きな原因の 1 つであると判明した．また，対象プロジェクト前後の開発実績を比較すると，充分な保守性向上の効果を確認できなかった．

対象プロジェクトの段階的再構築前後の開発実績の統計値を表 4.2 に示す．表 4.2 の案件数は，段階的再構築前では COBOL の基幹システム，段階的再構築後では切り出された新システムを対象にした保守開発案件のうち，製造規模，テスト規模，開発生産性が正しく記録されていた案件のみの数を示す．また，表 4.2 は，以下の平均値と中央値を示している．

**結合テスト 1 規模**　1 回目の結合テストでテスト対象となるソースコード規模．1 回目は 2 回目よりレベルが小さく，単体テストに近い．

**結合テスト 2 規模**　2 回目の結合テストでテスト対象となるソースコード規模．2 回

表 4.1　対象システムの概要

| | |
|---|---|
| システム概要 | 金融系基幹システム |
| システム区分 | バッチシステムおよびオンラインシステム |
| 言語区分 | オープン COBOL |

表 4.2　対象プロジェクトの段階的再構築前後の開発実績の統計値

| | | 段階的再構築前 | 段階的再構築後 | 増減率 |
|---|---|---|---|---|
| 案件数 | | 21 件 | 10 件 | - |
| 平均値 | 結合テスト 1 規模 [KLOC] | 22.75 | 21.24 | -7% |
| | 結合テスト 2 規模 [KLOC] | 24.04 | 23.28 | -3% |
| | 開発生産性 [KLOC/人月] | 0.27 | 0.19 | -30% |
| 中央値 | 結合テスト 1 規模 [KLOC] | 14.78 | 13.53 | -8% |
| | 結合テスト 2 規模 [KLOC] | 17.77 | 21.49 | +21% |
| | 開発生産性 [KLOC/人月] | 0.24 | 0.19 | -21% |

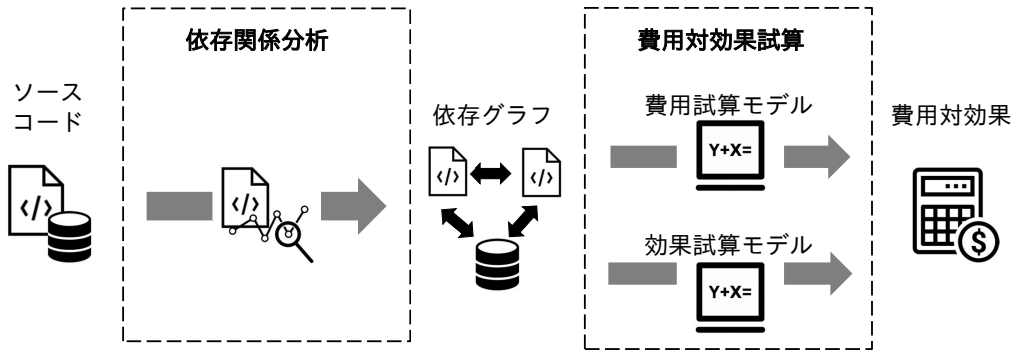


図 4.3　適用する費用対効果試算の概要

目は 1 回目よりレベルが大きく，システムテストに近い．

**開発生産性**　1 人月あたりの製造規模．単位は KLOC/人月．開発工数は基本設計からシステムテストまでの工数の合計．

結合テスト 1 規模と結合テスト 2 規模の値が小さいほど影響範囲が狭くなり，保守性が向上していることを示す．一方，開発生産性は値が大きいほど一人月あたりの実装行数が多くなり，保守性が向上していることを示す．表 4.2 の結合テスト 2 規模の中央値は段階的再構築の前後で増減率が正であるのに対し，開発生産性の平均値および中央値の増減率はどちらも負となっている．これらの結果は，段階的再構築が必ずしも保守性の向上に繋がったとは言えないことを示している．

### 4.3.2　適用の動機

対象プロジェクトにおいて，最終的に要した開発工数が計画段階で見積もった開発工数を超過したことや，段階的再構築による保守性向上への効果が充分でなかったことは，実際の開発現場における計画段階での段階的再構築の費用や効果を正確に見積もることが難しいことを示している．しかし，情報処理推進機構などの公開情報を調査しても，段階的再構築の開発事例が公開されていないため，類推法を適用できない．また，見積り手法やリファクタリングの費用対効果試算に関する既存研究はいくつかあるが [108, 109, 110, 111, 112]，段階的再構築における費用対効果試算の研究は我々の知る限り行われていない．そこで本研究では，現行システムのソースコードの依存関係分析を用いて費用対効果を試算する手法を採用した．

## 4.4　適用手法

本章では，4.3.1 節で説明した対象プロジェクトに適用する費用対効果を試算する手法（以降，本手法）について説明する．本手法は，4.2.3 節で述べたシステム化計画の

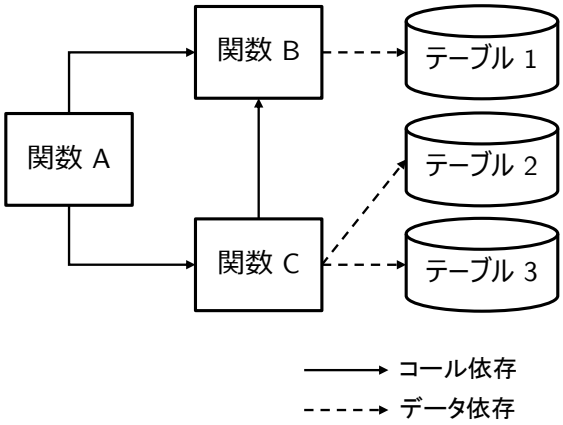


図 4.4　依存グラフの例

工程における試算として利用することを想定している．図 4.3 は，本手法の概要を示す．図 4.3 が示すように，本手法ではソースコードの依存関係分析に基づいてプログラム間の依存グラフを作成し，そのグラフをもとに段階的再構築の費用と効果を試算する．本章では最初に依存グラフについて説明し，費用試算，効果試算について説明する．

### 4.4.1　依存グラフ

まず本研究で使用する依存グラフについて述べる．本研究では，ソフトウェアの構成要素であるコードエンティティ（例：メソッドや関数など）やデータエンティティ（例：データベースのテーブルやグローバル変数など）をノードとし，ノード間の依存関係を辺（本研究では，辺はすべて有向辺）とするグラフ構造で表した依存グラフを使用する [119]．依存関係にはコール依存やデータ依存など様々な種類があり，この種類を本研究では「関係タイプ」と呼ぶ．

図 4.4 は本研究で使用する依存グラフの例を示している．この図では，ノードは関数とデータベースのテーブルを表し，辺はコール依存とデータ依存を表している．

### 4.4.2　費用試算

本節では，本手法における費用試算について説明する．費用試算では，段階的再構築にかかる開発工数を試算し，その結果に人月単価を掛け合わせることで金額に換算できる．

システム全体を一度で移行する場合，変更対象の規模に対して再開発の生産性を掛けることでパラメトリック法に従い開発工数を試算する [115]．しかし，著者は実際の開発現場で，システムの一部を切り出して段階的再構築する際，変更規模に生産性を

掛けただけでは，試算した工数が実績値より過少となる事例を経験した．そこで本研究では，新システム，旧システム，グルーコードの開発工数の合計によって段階的再構築の開発工数を求める手法を適用する．

新システムと旧システムの開発工数は，切り出し前のシステムの変更規模と開発生産性を掛けるパラメトリック法に基づいて計算される．一方で，グルーコードの開発工数を見積もるためには，依存グラフを用いる．ここでは，グルーコードの開発工数の試算モデルに絞り説明する．

**費用試算モデル**

ここでは，グルーコードの開発工数を算出するための費用試算モデルについて説明する．グルーコードの開発工数 $E$ は次の式で求める．

$$E = \alpha \times \sum_{d \in D} N_d L_d \tag{4.1}$$

式 (4.1) では，$\alpha$ は生産性（人月/規模），$D$ は依存関係のタイプの集合，$d$ は $D$ に含まれるある依存関係のタイプを意味する．また，$N_d$ は新システムと旧システムをまたがる依存関係 $d$ の本数，$L_d$ は依存関係 $d$ の 1 本あたりのコードの実装行数を意味しており，$N_d$ と $L_d$ を依存関係のタイプ $d$ ごとに合計し，生産性を掛けることでグルーコードの開発工数を求める．

式 (4.1) の $L_d$ と $\alpha$ は，旧システムと新システムのアーキテクチャにより変動する．そのため，これらの数値は過去の開発実績に基づいて試算するか，予備調査としてグルーコードを小規模に開発して開発工数を測定することによって試算する必要がある．

**費用試算の例**

図 4.5 は図 4.4 の例に対して段階的再構築を実施した後の関数とテーブルの例を示している．図 4.5 が示すように，関数 B が新システムへ切り出されたため，段階的再構築後は，関数 A と関数 B，関数 B と関数 C，関数 B とテーブル 1 の間の通信はすべてグルーコードを介して行われる．したがって段階的再構築には，新しいグルーコードの開発が必要である．表 4.3 は図 4.5 の費用試算した例を示している．この例では，

表 4.3　依存関係ごとの工数見積りの例

| 関係タイプ $d$ | $N_d$ | $L_d$ | 生産性 $\alpha$ | 開発工数 $E$ |
|---|---|---|---|---|
| コール | 20 | 40 | 855 | 0.93 |
| データ | 10 | 100 | 855 | 1.17 |
| 工数の合計 | | | | **2.10** |

**（注 1）** 依存関係ごとの LOC と生産性は暫定値．

**（注 2）** 生産性の単位は LOC/人月．開発工数の単位は人月．

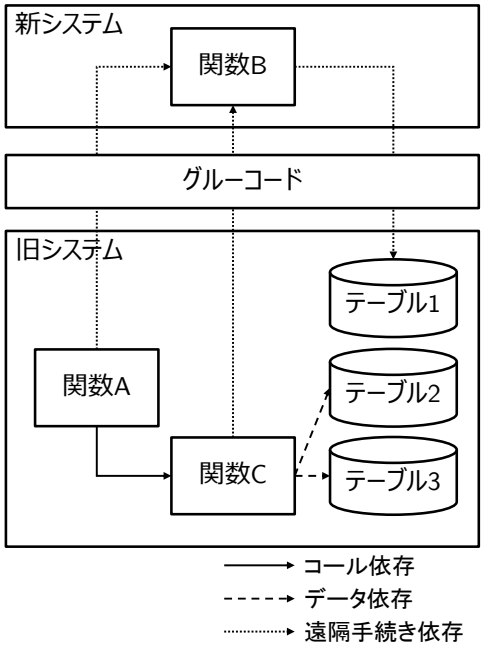


図 4.5　段階的再構築後の関数とテーブルの例

段階的再構築後にグルーコードを介した遠隔手続き依存に置き換えられるコール依存とデータ依存の数を，それぞれ 20 および 10 と仮定する．また，グルーコードで 1 つのコール依存を置き換えるために必要なソースコードの開発規模は 40 LOC，1 つのデータ依存の場合は 100 LOC と仮定する．したがって，表 4.3 に示されている依存関係の数と依存関係ごとの開発 LOC，生産性の逆数を掛けることで，コール依存のグルーコードの開発工数は 0.94 人月，データ依存のグルーコードの開発工数は 1.17 と試算できる．最後に，依存関係ごとのグルーコードの開発工数を合計して，グルーコード全体の開発工数は 2.11 人月と試算される．

### 4.4.3　効果試算

本節では，本手法における効果試算について説明する．本研究では段階的再構築後の保守開発工数の削減率を効果として試算する．モダナイゼーションにはさまざまな効果があるが，保守開発工数の削減率に着目する理由は，工数を減少することで開発アジリティが向上し，他の IT 分野への投資がしやすくなるためである．

本研究の効果試算では，影響範囲の削減率に基づいて保守開発工数の削減率を求める．ここで，影響範囲とは，一部のプログラムを変更した場合にその変更が他のプログラムにも影響を及ぼす範囲を意味する．影響範囲が広いほど，プログラムを理解しテストする範囲が広がるため，保守開発工数も増大する．一般に，ソフトウェア保守においてプログラム理解とテストにかかる工数割合が大きいため [120, 121, 122]，保

守開発工数の削減率は影響範囲の削減率から大きな影響を受ける [123, 124]．そこで本研究では，システムを切り出すことにより影響範囲を局所化できることから，それに伴う保守開発工数の削減率を段階的再構築の効果として見積もる．

**効果試算モデル**

ここでは，保守開発工数の削減率を求める効果試算モデルについて説明する．保守開発工数の削減率 $R$ は，次の式で求める．

$$R = 1 - \frac{\sum_{m \in M} P_m CI'_m}{\sum_{m \in M} P_m CI_m} \tag{4.2}$$

式 (4.2) では，$M$ はシステム全体のモジュール集合，$P_m$ はモジュール集合 $M$ に含まれるあるモジュール $m$ が 1 回の保守開発で変更される確率，$CI_m$ は段階的再構築前のモジュール $m$ の影響範囲，$CI'_m$ は段階的再構築後のモジュール $m$ の影響範囲を意味する．各モジュールに対する影響範囲は，そのモジュールが変更されたときにテストされるコード行数によって測定される．

式 (4.2) において，各モジュールが変更される確率とそのモジュールの影響範囲を掛け合わせた値を重み付き影響範囲という．そして，重み付き影響範囲の総和を取ることでシステム全体での影響範囲の期待値を計算する．実際のシステム開発において，1 回の保守開発ですべてのモジュールが平等に変更されることはない．また頻繁に変更されるモジュールは影響範囲を小さくする効果が高く，一方でめったに変更されないモジュールは影響範囲を小さくする効果が少ない．そのため，式 (4.2) では各モジュールの変更確率を考慮した重み付き影響範囲を基に効果を試算する．本研究では変更確率 $P(m)$ をモジュール $m$ の規模がシステム全体の規模の中で占める割合により算出する．これは，規模が大きいモジュールは変更される確率が高く，規模が小さいモジュールは変更される確率が低いという仮定に基づいている．さらに，影響範囲 $CI_m$ は，モジュール $m$ とモジュール $m$ に依存するすべてのモジュールのコード行数を合計することで計算される．これは，変更されたモジュールに依存するモジュールも変更の影響を受けるため，再テストが必要になるためである．本研究で提案する影響範囲の計算手法は既存研究 [110] の研究結果に基づいている．段階的再構築後の影響範囲の期待値は，新システムと旧システムをまたがる依存関係がなくなるものとして試算する．例えば，切り出し前は関数呼び出しで依存関係のあったモジュールが，切り出し後には遠隔手続き呼び出しに変わる場合，一般的にモジュール間の結合度が低下し，テストが容易になるため，影響範囲の期待値は小さくなる．

**効果試算の例**

図 4.6 は効果試算に用いる段階的再構築前後のモジュールの例を示している．また，表 4.4 と表 4.5 は，段階的再構築前後における影響範囲を示している．

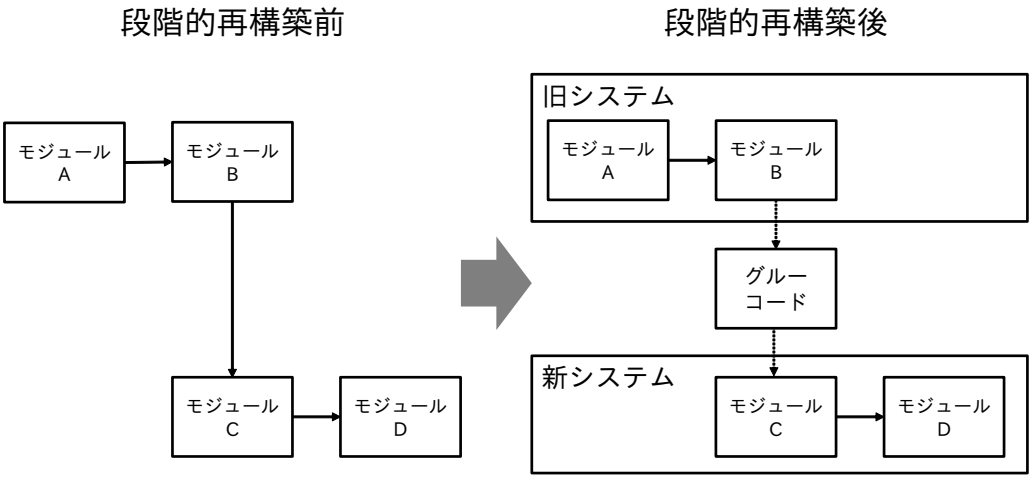


図 4.6　効果試算に用いるモジュールの例

最初に，段階的再構築前の影響範囲の期待値の算出例について説明する．この例では，図 4.6 の左側で示されたモジュール D が保守開発で変更される場合を考える．モジュール D が変更される確率 $P_D$ は，全モジュールのコード行数の合計に対するモジュール D のコード行数の割合から，40% と算出される．そしてモジュール D を変更した場合，依存しているモジュール A，B，C にも変更の影響が及ぶため，モジュー

表 4.4　段階的再構築前の影響範囲の例

| モジュール | LOC | $P_m$ | $CI_m$ | 重み付き影響範囲 |
|---|---|---|---|---|
| A | 10 | 10% | 10 | 1 |
| B | 20 | 20% | 30 | 6 |
| C | 30 | 30% | 60 | 18 |
| D | 40 | 40% | 100 | 40 |
| | | 影響範囲の推定値 | | 65 |

表 4.5　段階的再構築後の影響範囲の例

| モジュール | LOC | $P_m$ | $CI'_m$ | 重み付き影響範囲 |
|---|---|---|---|---|
| A | 10 | 10% | 10 | 1 |
| B | 20 | 20% | 30 | 6 |
| C | 30 | 30% | **30** | **9** |
| D | 40 | 40% | **70** | **28** |
| | | 影響範囲の期待値 | | **44** |

ル D の影響範囲 $CI_D$ はモジュール A，B，C，D のコード行数を合計して 100 となる．モジュール D は 40% の確率で変更され，変更された際の影響範囲は 100 であることから，重み付き影響範囲は 40 となる．モジュール D 以外についても同様に重み付き影響範囲を算出して総和をとることで，段階的再構築前の影響範囲の期待値は 65 となる．

次に，段階的再構築後にモジュール C と D が新システムとして切り出され，グルーコードによってモジュール B と C が連携された場合の影響範囲の期待値の算出例について説明する．この例では，図 4.6 の右側で示されたモジュール D が保守開発で変更される場合を考える．まず，段階的再構築前と同様に，モジュール D の変更確率 $P_D$ は 40% となる．しかし，モジュール D の影響範囲 $CI'_D$ は 100 から 70 に減少する．その理由は，モジュール D が変更された場合，モジュール D 自身と C には段階的再構築前と同様に変更の影響が及ぶが，グルーコードによってモジュール C と B の間の依存関係が解消されたため，モジュール B および A には変更の影響が及ばないためである．これにより，モジュール D の重み付き影響範囲は 40 から 28 に減少する．また，モジュール C の影響範囲 $CI'_C$ も 60 から 30 に減少し，重み付き影響範囲は 18 から 9 に減少する．一方でモジュール A と B については，段階的再構築前後で影響範囲は変化しない．この結果，全モジュールの重み付き影響範囲の総和を取ると，段階的再構築後の影響範囲の期待値は 65 から 44 に減少し，保守開発工数の削減率は 32% に達すると推定する．

## 4.5 調査

### 4.5.1 調査目的とリサーチクエスチョン

本調査では，4.3.1 節で説明した対象システムを対象に，段階的再構築において 4.4 章で説明した本手法の妥当性を調査する．

そこで，本調査では次の 2 つのリサーチクエスチョンを設定した．

RQ1　費用試算結果と実績値の乖離はどれくらいか？
RQ2　効果試算結果と実績値の乖離はどれくらいか？

この 2 つのリサーチクエスチョンを解くことで，費用試算モデルと効果試算モデルの妥当性を評価する．

さらに，プロジェクト関係者にヒアリングを行うことで，提案した試算モデルおよび試算結果に対する定性的評価を行う．

### 4.5.2 調査 1：費用試算の妥当性

**調査方法**

RQ1 に回答するため，本調査では 4.4.2 節で説明した費用試算モデルを対象システムに適用し，試算した開発規模と実績値との乖離を比較した．

まず，対象システムに対して依存関係分析を行い，依存グラフの頂点はファイル，辺は呼び出し関係である依存グラフを作成する．COBOL プログラムでは，ファイル単位でプログラムにコンパイルされ，CALL 文を使って他プログラムを呼び出す．

次に，新システムとして切り出して段階的再構築するプログラムと，旧システムとして残すプログラムを分類する．対象プロジェクトにおいて実際に切り出された COBOL プログラムを，本研究でも切り出し対象として選定した．そして，新システムとして切り出し対象のプログラムと，旧システムに残すプログラム間で跨る依存関係の本数を依存関係グラフに基づいて集計した．

最後に，新システム，旧システム，グルーコードの開発規模をそれぞれ計算し，それらを合計することで，システム全体の開発規模を算出した．新システムおよび旧システムの開発規模は，対象プロジェクトの計画段階において試算された値を再利用した．ただし，計画段階ではグルーコードの開発が必要になるとは想定されていなかったため，新システムの旧システムの開発規模のみが試算対象であった．グルーコードの開発工数は，4.4.2 節で述べた式 (4.1) に基づき算出した．ただし，本調査は工数見積りではなく規模見積りで評価するため，生産性 $\alpha$ は計算に含めない．

本調査では，実際の開発規模（3,691KLOC）と，費用試算結果との乖離を定量的に比較した．さらに，計画段階での試算値（2,675KLOC）と比較し，費用試算モデルの妥当性を調査した．

費用試算結果の乖離率は次の式で求める．

$$\text{乖離率} = \frac{\text{試算値} - \text{実績値}}{\text{実績値}} \tag{4.3}$$

**調査結果**

対象システムでは，新システムから旧システムへの呼び出し関係は 3,431 個，旧システムから新システムへのよびだし関係は 80 個であった．したがってシステム間を跨る呼び出し関係は 3,511 個となる．依存関係 1 本あたりのグルーコードの実装行数は，システム内の 1 ファイルあたりの平均 LOC から算出し，343.3LOC だった．これらの結果から，グルーコードの開発規模は，1,205LOC と試算できる．この試算結果と，グルーコードを含んでいない計画段階の試算 2,675KLOC を合計することで，システム全体の開発規模は 3,880KLOC という試算結果が得られた．

対象プロジェクトの開発規模として，費用試算モデルの試算結果，対象プロジェク

トの計画段階の試算結果および実績値を表 4.6 に示す．この表から費用試算モデルの試算値は 3,880KLOC と算出され，実際の開発規模の実績値である 3,691KLOC と比較して，乖離率が 5.1% であったことがわかる．一方で対象プロジェクトの計画段階の試算値と実績値との乖離が-27.5% である．これらの結果から，費用試算モデルを用いることで，実績値との乖離が小さく，より正確に費用を試算できることが示唆される．これは本手法における費用試算がグルーコードの開発規模を考慮しているためと考えられる．

RQ1 への回答

本手法の費用試算結果と実績値の乖離は +5.1% で，費用試算モデルを使用することで対象プロジェクト計画段階の試算結果より乖離の少ない試算ができた．

### 4.5.3 調査 2：効果試算の妥当性

**調査方法**

RQ2 に回答するため，本調査では 4.4.3 節で説明した効果試算モデルを対象システムに適用し，試算したテスト規模の削減量と実績値との乖離を比較した．テスト工程は開発工程の中で大きな割合を占め，開発工程全体に大きな影響を与えるため，本調査ではテスト規模に比例してテスト工程の工数も増減するという仮定のもと，テスト規模の削減量を試算する．

表 4.6　費用試算：対象プロジェクトの開発規模

| | 値 | 乖離率 |
|---|---|---|
| 費用試算モデルの試算値 | 3,880KLOC | +5.1% |
| 計画段階の試算値 | 2,675KLOC | -27.5% |
| 実績値 | 3,691KLOC | - |

表 4.7　効果試算：段階的再構築前後の比較

| | | 段階的再構築後の実績値 | 効果試算モデルの試算値 | 乖離率 |
|---|---|---|---|---|
| 平均値 | 結合テスト 1 規模 [KLOC] | 21.24 | 19.79 | -7% |
| | 結合テスト 2 規模 [KLOC] | 23.28 | 20.91 | -10% |
| | 開発生産性 [KLOC/人月] | 0.19 | 0.32 | +68% |
| 中央値 | 結合テスト 1 規模 [KLOC] | 13.53 | 12.86 | -5% |
| | 結合テスト 2 規模 [KLOC] | 21.49 | 15.46 | -28% |
| | 開発生産性 [KLOC/人月] | 0.19 | 0.27 | +42% |

まず，調査 1 と同様に，対象システムの依存関係を分析して依存グラフを作成し，新システムとして切り出して段階的再構築するプログラムと，旧システムとして残すプログラムに分類する．その後，4.4.3 節で述べた式 (4.2) を用いて，段階的再構築前後の保守開発工数の削減率を算出した．

次に，表 4.2 で示した対象プロジェクトの段階的再構築前の開発実績の統計値に対して，上記で算出した削減率を乗算し，段階的再構築後の値を試算する．最後に，段階的再構築後の各開発実績と試算結果を比較し，本手法による試算結果の乖離を評価する．効果試算結果の乖離率も，費用試算の乖離率と同様の式で求める．

**調査結果**

対象システムでは，段階的再構築前の影響範囲の期待値（$\sum_{m \in M} P_m CI_m$）は 7.99KLOC, 段階的再構築後の影響範囲の期待値（$\sum_{m \in M} P_m CI'_m$）は 6.99KLOC であった．この結果から, 式 (4.2) より, 保守開発工数の削減率 $R = 1-(6.99/7.99) = 0.125$ となり，13% 削減可能という結果が得られた．

段階的再構築後の実績値と，段階的再構築前の実績値に対して 13% 削減した試算結果の比較を表 4.7 に示す．この表から，効果試算の結果は，段階的再構築後の実績値と比較して-28%〜+68% 乖離していることがわかる．乖離の程度はさまざまであるが，テスト規模は実績値が大きく，開発生産性は実績値が小さいため，すべてにおいて試算結果ほど効果を得ることはできなかった．

表 4.2 において段階的再構築後にテスト規模が小さくなった結合テスト 1（平均値と中央値）および結合テスト 2（平均値）では，段階的再構築後の実績値と効果試算の結果の乖離率が-10%〜-5% と小さかった．一方，段階的再構築後にテスト規模が大きくなった結合テスト 2（中央値）では，乖離率が-28% と乖離が大きくなる傾向が見られた．これにより，段階的再構築前後の開発実績の増減が，効果試算の結果との乖離に影響を与えている可能性が考えられる．

また，段階的再構築後に開発生産性が悪化した理由として，次の 2 つの要因が考えられる．1 つ目の要因は，言語の違いである．対象プロジェクトでは，COBOL から Java に再構築されたが，言語の違いによりテスト規模や開発生産性の値が悪化した可能性がある．テスト規模および開発生産性とは異なる尺度を用いることで，保守性向上効果を定量的に可視化できる可能性がある．2 つ目の要因は，段階的再構築の中で機能追加が行われたことである．モダナイゼーションプロジェクトの期間が長くなると，その間のビジネス状況の変化も大きくなり，その変化に対応するために機能追加が入る傾向がある．段階的再構築の中での機能追加は，システムの保守性を悪化させ，開発生産性の悪化に繋がっている可能性がある．

RQ2 への回答

本手法の効果試算結果と実績値の乖離は-28%〜+68% となった．ただし一部比較項目において，段階的再構築前後の開発実績が悪化しており，テスト規模削減および生産性向上が前提である効果試算モデルの正確性の評価が難しい．

### 4.5.4 プロジェクト関係者ヒアリング

本調査では，5.2 節と 5.3 節で定量的に評価した費用試算モデルおよび効果試算モデルの妥当性を定性評価するため，対象プロジェクトの関係者に対して次の 3 つのヒアリング調査を実施した．

**質問 1**　試算結果は妥当感があるか
**質問 2**　試算モデルは納得感があるか
**質問 3**　試算モデルは今後プロジェクトに適用可能か

質問 1「試算結果は妥当感があるか」では，4.5.2 節で説明した調査 1 と 4.5.3 節で説明した調査 2 の試算結果を関係者に説明することで，試算結果の妥当性を調査した．質問 2「試算モデルは納得感があるか」では，式 (4.1) と式 (4.2) を関係者に説明し，これらの試算モデルが実際のシステムの関係者にとって納得できるものであるかを調査した．最後に質問 3「試算モデルは今後プロジェクトに適用可能か」では，説明した試算結果と試算モデルをふまえて，今後プロジェクトにこの試算モデルを適用可能か（適用したいと思うか）調査した．

本調査は企業でモダナイゼーションの経験を持つプロジェクトマネージャー，PMO エキスパート，開発者の計 3 名を対象に調査を実施した．弊社では，ユーザー企業の経営層に対してモダナイゼーションの費用対効果を説明し，経営層から合意を得る活動を継続的に行っている．本調査においては，費用対効果の説明責任を持ったプロジェクトマネージャーと，実際に試算したメンバーに対しヒアリング実施した．これらの回答者には，ユーザー企業の立場を踏まえた観点から回答してもらうよう依頼した．

回答形式は，各質問に対して費用試算モデルと効果試算モデルのそれぞれについて 5 段階評価を行い，その理由を自由記述形式で回答してもらった．

#### 調査結果

プロジェクト関係者ヒアリングの調査結果を表 4.8 に示す．この表で示されているように，質問 1「試算結果は妥当感があるか」については，費用試算は「妥当」,「やや妥当」,「どちらでもない」と回答した人が 1 人ずつおり，妥当と感じた人の方が比較的に多かった．自由記述のコメントによると，「製造規模だけでなくグルーコード規模を含めて試算することに対して，納得感を感じる」という意見があった．一方，効果試算については，3 人のうち 2 人が「どちらでもない」，1 人が「やや妥当でない」と

回答し，妥当性を感じていない人が比較的に多いという結果となった．自由記述のコメントによると，「効果試算モデルではソースコードに対しての変更影響のみを考慮しているが，ソースコードだけでなくデータベーステーブルへの変更影響も考慮したほうが，効果の体感と近くなるのではないか」という意見があった．

次に，質問 2「試算モデルは納得感があるか」という質問については，費用試算では 3 人のうち 1 人が「納得感がある」，2 人が「どちらでもない」と回答した．自由記述のコメントでは，「システム間を連携するために必要な部品の見積り式としての納得感があるが，部品の必要性は一般的ではなくシステム特性に依存する」という意見があった．また効果試算については，3 人のうち 1 人が「やや納得感がある」，2 人が「どちらでもない」と回答した．自由記述のコメントでは，「プロジェクト特性や適用可能な工程など，前提条件があるように思うことと一般性が低い」という意見があった．費用試算と効果試算の双方において，中立的な意見が多数を占める一方，若干の納得傾向が確認できた．ただし，どちらの式も理解が難しいという意見があった．

最後に，質問 3「試算モデルは今後プロジェクトに適用可能か」という質問については，費用試算，効果試算の双方において，肯定的な意見と否定的な意見に分かれた．費用試算ついては，3 人のうち 1 人が「適用できる」，1 人が「やや適用できる」，1 人が「やや適用が難しい」と回答した．肯定的な意見として，「自身が担当するプロジェクトをもとにした試算モデルのため適用しやすい」という意見が得られた．一方で否

表 4.8　プロジェクト関係者ヒアリングの結果

| 質問 | 回答: 費用試算　（人） | | 回答: 効果試算　（人） | |
|---|---|---|---|---|
| 質問 1: 試算結果は妥当感があるか？ | 妥当 | 1 | 妥当 | 0 |
| | やや妥当 | 1 | やや妥当 | 0 |
| | どちらでもない | 1 | どちらでもない | 2 |
| | やや妥当でない | 0 | やや妥当でない | 1 |
| | 妥当でない | 0 | 妥当でない | 0 |
| 質問 2: 試算モデルは納得感があるか？ | 納得感がある | 1 | 納得感がある | 0 |
| | やや納得感がある | 0 | やや納得感がある | 1 |
| | どちらでもない | 2 | どちらでもない | 2 |
| | やや納得感がない | 0 | やや納得感がない | 0 |
| | 納得感がない | 0 | 納得感がない | 0 |
| 質問 3: 試算モデルは今後プロジェクトに適用可能か？ | 適用できる | 1 | 適用できる | 1 |
| | やや適用できる | 1 | やや適用できる | 0 |
| | どちらでもない | 0 | どちらでもない | 1 |
| | やや適用が難しい | 1 | やや適用が難しい | 1 |
| | 適用が難しい | 0 | 適用が難しい | 0 |

定的な意見として，「対象プロジェクトで発生した問題はグルーコードだけではなく，この試算モデルがカバーしない範囲の検討の必要」という懸念点が指摘された．効果試算ついては，3 人のうち 1 人が「適用できる」，1 人が「どちらでもない」，1 人が「やや適用が難しい」と回答した．肯定的な意見として，「次のプロジェクトの刷新時には，この試算モデルも活用しつつ効果を定量評価して顧客に訴求したい」という意見が得られた一方で，否定的な意見として，「効果は生産性向上だけではなく，品質，費用，納期のバランスが重要であること．この試算だけでは顧客満足に繋がりにくい」という点が指摘された．

プロジェクト関係者ヒアリングの結論

試算結果の妥当感については，費用試算は比較的肯定的，効果試算は比較的否定的な傾向がみられた．試算モデルの納得感については，費用試算と効果試算ともに中立的な意見が多かった．試算モデルの今後の適用可能性については，費用試算と効果試算ともに肯定的な意見と否定的な意見に分かれた．

### 4.5.5 妥当性への脅威

費用試算において，本調査では依存関係 1 本あたりの実装行数を，プログラム 1 個あたりの平均行数から試算した．実際には事前検証などを実施することで依存関係 1 本あたりの実装行数をより精緻に求めることができる．しかし，一般的な実開発において計画段階で事前検証の実施は難しい．表 4.6 より，本実験の計算方法でも計画段階の試算より乖離率が小さいことから，プログラム 1 個あたりの平均行数を使った試算精度であっても許容できる．また，本調査では工数見積りではなく規模見積りで評価したため，生産性 $\alpha$ は計算に含めていない．ただし，生産性 $\alpha$ はプロジェクトごとに定められる定数値であり，対象プロジェクトにおいても生産性ではなく規模の見積り誤差に起因して工数見積りが乖離したため，本調査では規模見積りで評価した．

効果試算において，乖離率の閾値を決める基準値がないため，効果試算結果が正確であると判断できる乖離率の上限値および下限値はない．したがって，効果試算モデルが正確であるか否かの判断が難しい．

さらに，見積りの正確性の評価指標として乖離率以外の指標も存在する．本研究では実績値との乖離率をもって調査の結論としたが，別の指標を採用することでことなる結果となる可能性がある．

また本調査では，段階的再構築によりテストされるコード行数が減少し，開発生産性が向上するという前提のもと，保守性向上効果を試算した．しかし，表 4.2 より，結合テスト 2 規模（中央値）の増減率が正に，また開発生産性（平均値と中央値）の増減率が負になっており，むしろ段階的再構築前より悪化している．したがって，段階的再構築によりテスト規模の削減および保守開発工数の削減効果を得られない懸念が

ある．段階的再構築の別の効果として，修正が必要なテーブル数の削減による保守性の向上がある．本調査とは別の依存タイプを分析し，新たな前提をもとに検証することは今後の課題である．

本調査では，弊社でモダナイゼーションの経験を有するメンバーを対象にヒアリングを実施した．しかしながら，弊社はベンダー企業であるため，ユーザー企業の視点に基づく直接の回答を得ることはできなかった．費用対効果の見積りは，ユーザー企業の経営層が納得できるか否かが重要である．したがって，ユーザー企業を対象にヒアリングを実施することで，より客観的な評価を得られる可能性がある．しかし，弊社はこれまで，ユーザー企業の経営層に対してモダナイゼーションの費用対効果を説明し，経営層から合意を得る活動を継続的に行っている．本調査においては，費用対効果の説明責任を担うプロジェクトマネージャーと，実際に試算したメンバーに対しヒアリングを実施した．これらの回答者には，ユーザー企業の立場を踏まえた観点から回答してもらうよう依頼している．そのため，本調査はユーザー企業の視点を一定程度反映していると考えられる．

## 4.6 関連研究

Leitch らは，リファクタリングにおける依存関係分析を用いて費用対効果を試算する手法を提案した [110]．Leitch らの手法では，制御依存とデータ依存の依存関係からリファクタリングにより削減可能なシステムの保守開発工数を試算する．また COCOMO II を用いてリファクタリングの実施工数を試算する．しかし，Leitch らの手法ではシステム内のリファクタリングにおける費用対効果の算出しかできず，段階的再構築の費用対効果を試算する点で適用手法と異なる．

Cui らは，依存関係の修正方法と修正作業工数の相関関係を調査した [109]．これにより，依存関係の種類ごとに修正作業工数を見積もることができる．ただし，依存関係の修正方法にシステム切り出しがない点において，適用手法の費用試算とは異なる．

Rebêl らは，アスペクト指向切り出しにより，オブジェクト指向設計と比較して，設計上の安定性と保守作業の関係性を調査した [111]．これにより，アスペクト指向切り出しによる保守開発工数の削減効果を試算できる．ただし，アスペクト指向切り出しではなく，別システムに切り出す点で，適用手法の効果試算とは異なる．

Cai らは，ソースコード，アーキテクチャ情報，バージョン履歴をもとに，リファクタリングの費用対効果を判断するフレームワークを開発した [108]．費用対効果を判断するフレームワークとして，提案手法と類似している．一方で具体的な費用を試算する手法は示されていない．また効果試算は影響範囲に着目しておらず，過去のリファクタリング作業との相関関係により試算する点で，適用手法とは異なる．

Xiao らは，アーキテクチャの技術的負債と保守開発工数との相関関係を調査した [112]．これにより，技術的負債の存在による保守開発工数の増加量を試算できる．た

だし，システム切り出しによって保守開発工数の削減量を試算しない点で，適用手法とは異なる．

小林らは，システム障害予測の精度向上を目的として，変更の影響波及量を定量化するメトリクスを提案した [119]．依存グラフを用いて変更の影響範囲を見極める点で，提案手法と類似しているが，障害予測の精度向上を目的としており開発工数の削減量試算を目的としていない点で，適用手法と異なる．

早瀬らは，保守開発作業の労力見積りに用いることを目的として，影響波及解析を使ったメトリクスを提案した [125]．依存グラフを用いて変更の影響範囲を見極める点で，提案手法と類似している．しかし早瀬らの手法は，特定のプログラムの保守開発工数の試算のみ実施しており，システム全体での保守開発工数の削減率を試算する適用手法と異なる．

## 4.7 まとめと今後の課題

段階的再構築はモダナイゼーション失敗リスクを軽減できる一方，計画段階で段階的再構築の費用対効果を正確に見積もることは困難である．実際に第 1 著者および第 4 著者が所属する企業で段階的再構築を実施した事例では，計画段階で見積もった値と実績値の乖離が発生していた．そこで本研究では，過去に段階的再構築を実施した大規模な金融システムに対して，依存関係分析を用いた費用対効果試算を適用し，実績値との乖離率を比較した．また，プロジェクトの関係者にヒアリングを行い，適用手法の妥当性を定性評価した．評価の結果，費用試算は実績値と乖離率が小さかったのに対し，効果試算は実績値との乖離が大きかった．プロジェクト関係者ヒアリングの結果からは，費用試算結果はやや妥当だが，効果試算結果はやや妥当でないという意見が得られた．また費用試算と効果試算ともに，試算モデルの納得感については中立的な意見が多く，今後の適用可能性については肯定派と否定派に分かれた．

今後の課題として，以下の 3 点が挙げられる．

**評価結果の信頼性向上**　現時点では 1 つの案件でのみ評価を行なった．評価対象のシステムを増やして交差検証を行うなど，評価結果の信頼性を向上させることは今後の課題である．

**依存関係の拡充**　本研究ではプログラムの呼び出し関係のみを依存関係として解析した．他にもデータベースアクセスなど，依存関係のタイプは存在する．タイプの種類を増やし，新たなタイプをもとにした試算結果の精度検証は今後の課題である．

**他言語への適用**　本研究では，COBOL 言語を対象に試算を行った．COBOL 以外の言語でも提案手法を適用して試算結果が得られるのか，今後の課題である．

# 第 5 章

# おわりに

## 5.1 まとめ

本論文では，コードクローンの把握を支援する目的で 2 つの研究を，モダナイゼーションの費用対効果の試算を支援する目的で 1 つの研究を実施した．

1. コードクローンの把握を支援
   （a）情報検索技術に基づく細粒度ブロッククローン検出
   （b）情報検索技術と深層学習を用いたコード片類似性判定法の比較調査
2. モダナイゼーションの費用対効果の試算を支援
   （a）段階的再構築における依存関係分析を用いた費用対効果の試算

1-(a) については，情報検索技術の一種である TF-IDF と LSH を利用したブロッククローン検出手法を提案した．本手法では，構文解析によりコードブロックを抽出し，TF-IDF を利用してコードブロックを特徴ベクトルに変換した後，特徴ベクトル間の類似度を計算することで，意味的に処理が類似したブロッククローンを検出する．また，クラスタリング手法の一種である LSH を用いることで，高速なブロッククローン検出を実現した．これにより，既存のコードクローン検出手法よりも高精度でコードクローンを検出し，さらに保守作業を行いやすいコードクローンを検出が可能となった．また，クラスタリング手法や特徴ベクトルのデータ構造を工夫することで，既存手法より高速かつ低メモリ消費での検出が可能となった．

1-(b) については，コード片の処理内容の意味的な類似性を高精度かつ高速に判定するため，情報検索技術と深層学習の効果的な組み合わせを調査した．調査では，5 種類のベクトル表現間での比較と，深層学習を用いた既存手法との比較を実施した．調査の結果，情報検索技術に基づくベクトル表現として LSI（Latent Semantic Indexing）を用いた手法が，最も高精度かつ高速であることが明らかになった．また，既存手法である DeepSim と比較して，LSI（Latent Semantic Indexing）を用いた手法は再現率，適合率，F 値のいずれも高く，学習時間においても約 350 倍高速であることが確

認できた．

2-(a) については，過去に段階的再構築を実施した大規模な金融システムを対象に，依存関係分析を用いた費用対効果試算手法を適用し，その妥当性を検証した．本研究では，システムの規模と依存関係の情報を活用し，費用として段階的再構築に必要な工数を，効果として削減可能な保守開発工数を試算し，実績値との乖離度合いを比較した．また，プロジェクトの有識者にヒアリングを行い，適用手法の妥当性を定性評価した．評価の結果，費用試算は実績値との乖離が小さい一方で，効果試算は実績値との乖離が大きいことが明らかになった．有識者ヒアリングの結果からは，費用試算結果については「やや妥当」との意見が得られたものの，効果試算結果については「やや妥当でない」との意見が多かった．また，試算モデルの納得感については中立的な意見が多く，今後の適用可能性については肯定派と否定派に分かれる結果となった．

## 5.2 今後の研究方針

今後，本論文で述べた研究成果を応用し，ソフトウェア保守およびモダナイゼーションの作業をより容易なものにしていきたいと考えている．具体的には，本論文で提案したコードクローン検出手法とモダナイゼーションの費用対効果試算手法を実際の大規模プロジェクトに適用し，開発者からフィードバックを得ることで，手法の有用性を評価したい．

さらに，本論文で扱ったコードクローン検出およびモダナイゼーションの研究分野において，大規模言語モデル（以降，LLM という）の活用を試みたいと考えている．LLM は主に自然言語処理の分野で高い成果を上げており，現在ではソフトウェア工学の研究分野においても注目を集めている [126, 127]．

コードクローン検出の分野では，構文的な類似度の低いコードクローンを高精度で検出することが課題として残されている [16]．本論文では，情報検索技術と深層学習を組み合わせることで，既存研究より高速かつ高精度に構文的な類似度の低いコードクローンを検出できることを示した．近年では，LLM を用いたコードクローン検出手法も提案されており，従来の手法よりも高い精度での検出を実現している [128]．しかし，構文的な類似度の低いコードクローンに対する検出精度はまだ十分とはいえず，さらなる改善の余地がある．そこで，情報検索技術や深層学習に加えて LLM を組み合わせることで，より高精度で高速なコードクローンの検出手法を提案したいと考えている．

またモダナイゼーションの分野では，レガシーシステムの知識不足の課題が指摘されている [18]．本論文では，モダナイゼーションのシステム化計画工程における費用対効果の見積りの難しさに着目し，段階的再構築の費用対効果試算について研究した．システム化計画を終え，次の要件定義以降の工程に進む際には，レガシーシステムに関する知識不足という課題に直面することが多い．この課題は，レガシーシステム特

有の技術的な専門知識のみならず，ドキュメント不足など起因して欠落してしまった業務知識にも及ぶ．これらの知識不足は，モダナイゼーションを進める上で重要な課題である．近年，LLM を活用してレガシーシステムの知識不足を解消する研究や取り組みが進められている [129, 130, 131]．今後，ソースコード静的解析と LLM を活用し，ソースコードから設計に関する抽象概念を復元するリバースエンジニアリング手法を提案したいと考えている．

# 参考文献


[1] IEEE Std 1219, Standard for Software Maintenance.

[2] ISO/IEC 14764:2006, Software Engineering — Software Life Cycle Processes — Maintenance.

[3] JIS X 0161:2008, ソフトウェア技術 – ソフトウェアライフサイクルプロセス – 保守.

[4] M. Page-Jones, The practical guide to structured systems design, Yourdon Press, 1988.

[5] S. Yip, T. Lam, A software maintenance survey, in: Proceedings of the 1st Asia-Pacific Software Engineering Conference, 1994, pp. 70–79.

[6] A. April, A. Abran, Software Maintenance Management: Evaluation and Continuous Improvement, John Wiley & Sons, 2008.

[7] 日本情報システム・ユーザー協会, 企業 IT 動向調査報告書 2019, 2019.

[8] 肥後芳樹, 楠本真二, 井上克郎, コードクローン検出とその関連技術, 電子情報通信学会論文誌 J91-D (6) (2008) 1465–1481.

[9] K. Inoue, C. K. Roy, Code Clone Analysis: Research, Tools, and Practices, Springer, 2021.

[10] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming 74 (7) (2009) 470–495.

[11] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎, 佐野建樹, コードクローン変更管理システムの開発と実プロジェクトへの適用, 情報処理学会論文誌 54 (2) (2013) 883–893.

[12] 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 31 (1) (2014) 14–29.

[13] 三木聡, 大歳始, 浅原明広, 大澤俊晴, 千葉滋, 実務で使われるコードクローン検出・追跡システムをめざして, in: 日本ソフトウェア科学会第 40 回大会講演論文集, 2023.

[14] 佐野真夢, 吉田則裕, 春名修介, 井上克郎, 情報検索技術に基づく関数クローン

検出を用いた変更管理システムの開発, 情報処理学会研究報告 2015-SE-190 (4) (2015) 1–8.

[15] H. Wei, M. Li, Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code, in: Proceedings of the 26th International Joint Conference on Artificial Intelligence, 2017, pp. 3034–3040.

[16] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, B. Maqbool, A systematic review on code clone detection, IEEE Access 7 (2019) 86121–86144.

[17] K. Bennett, Legacy systems: Coping with success, IEEE Software 12 (1) (1995) 19–23.

[18] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, J. Hage, How do professionals perceive legacy systems and software modernization?, in: Proceedings of the 36th International Conference on Software Engineering, 2014, p. 36–47.

[19] E. Arranga, F. Coyle, Cobol: perception and reality, Computer 30 (3) (1997) 126–128.

[20] N. Veerman, Revitalizing modifiability of legacy assets, Journal of Software Maintenance and Evolution: Research and Practice 16 (4-5) (2004) 219–254.

[21] MarketsandMarkets, Application Modernization Services Market by Service Type (Cloud Application Migration, Application Re-Platforming, Post Modernization), Application Type (Legacy, Cloud-hosted, Cloud-native) - Global Forecast to 2029.

[22] R. C. seacord, D. Plakosh, G. A. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices, Addison-Wesley Longman Publishing Co., Inc., 2003.

[23] 独立行政法人情報処理推進機構, ソフトウェア開発データ白書 2018-2019 (2018).

[24] E. Chikofsky, J. Cross, Reverse engineering and design recovery: a taxonomy, IEEE Software 7 (1) (1990) 13–17.

[25] Imagix Corporation, Imagix 4D, `http://www.imagix.com/products/products.html`.

[26] IBM, Rational software modeler, `http://www-01.ibm.com/software/awdtools/modeler/swmodeler/`.

[27] T. Biggerstaff, Design recovery for maintenance and reuse, Computer 22 (7) (1989) 36–49.

[28] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[29] N. Shi, R. A. Olsson, Reverse engineering of design patterns from Java source code, in: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, 2006, p. 123–134.

[30] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, Design pattern detection using similarity scoring, IEEE Trans.Softw. Eng. 32 (11) (2006) 896–909.

[31] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, W. Tichy, Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance, IEEE Trans.Softw. Eng. 28 (6) (2002) 595–606.

[32] X. Ren, B. Ryder, M. Stoerzer, F. Tip, Chianti: a change impact analysis tool for java programs, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004, pp. 432–448.

[33] G. Rothermel, M. Harrold, A safe, efficient regression test selection technique, ACM Trans. Softw. Eng. Methodol. 6 (2) (1997) 173–210.

[34] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.

[35] E. J. Weyuker, Evaluating software complexity measures, IEEE Trans. Softw. Eng. 14 (9) (1988) 1357–1365.

[36] V. Basili, L. Briand, W. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Trans. Softw. Eng. 22 (10) (1996) 751–761.

[37] M. Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering, 1981, p. 439–449.

[38] T. M. Meyers, D. Binkley, An empirical study of slice-based cohesion and coupling metrics, ACM Trans. Softw. Eng. Methodol. 17 (1).

[39] Y. Kataoka, T. Imai, H. Andou, T. Fukaya, A quantitative evaluation of maintainability enhancement by refactoring, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 576–585.

[40] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: improving the design of existing code, Addison-Wesley Longman Publishing Co., Inc., 1999.

[41] W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).

[42] D. Rattan, R. Bhatia, M. Singh, Software clone detection: A systematic review, Information and Software Technology 55 (7) (2013) 1165–1199.

[43] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer

Programming 74 (7) (2009) 470–495.

[44] R. Baeza-Yates, B. Ribeiro-Neto, Modern information retrieval: The concepts and technology behind search, Addison-Wesley, 2011.

[45] 北. 研二, 津. 和彦, 獅. 堀正幹, 情報検索アルゴリズム, 共立出版, 2002.

[46] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎, 情報検索技術に基づく高速な関数クローン検出, 情報処理学会論文誌 55 (10) (2014) 2245–2255.

[47] M. L. Brodie, M. Stonebraker, Darwin: On the incremental migration of legacy information systems, Tech. rep., GTE Laboratories, Inc (1993).

[48] S. Comella-Dorda, G. Lewis, P. Place, D. Plakosh, R. Seacord, Incremental modernization of legacy systems, Tech. rep., Carnegie Mellon University, Software Engineering Institute's Digital Library (2001).

[49] 独立行政法人情報処理推進機構, 共通フレーム 2013, SEC BOOKS, 2013.

[50] 独立行政法人情報処理推進機構, 経営者が参画する要求品質の確保, SEC BOOKS, 2006.

[51] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the International Conference on Software Maintenance, 1998, pp. 368–377.

[52] P. Indyk, R. Motwani, Approximate nearest neighbors: towards removing the curse of dimensionality, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, 1998, pp. 604–613.

[53] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, T. Sano, Applying clone change notification system into an industrial development process, in: Proceedings of the 21th IEEE International Conference on Program Comprehension, 2013, pp. 199–206.

[54] Q. Lv, W. Josephson, Z. Wang, M. Charikar, K. Li, Multi-probe lsh: efficient indexing for high-dimensional similarity search, in: Proceedings of the 33rd International Conference on Very Large Data Bases, 2007, pp. 950–961.

[55] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, L. Schmidt, Practical and optimal lsh for angular distance, in: Proceedings of the 28th International Conference on Neural Information Processing Systems, 2015, pp. 1225–1233.

[56] K. Terasawa, Y. Tanaka, Spherical lsh for approximate nearest neighbor search on unit hypersphere, in: Proceedings of the 10th International Conference on Algorithms and Data Structures, 2007, pp. 27–38.

[57] 古賀久志, ハッシュを用いた類似検索技術とその応用, 電子情報通信学会 基礎・境界ソサイエティ Fundamentals Review 7 (3) (2014) 256–268.

[58] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: a multilinguistic token-based

code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28 (7) (2002) 654–670.

[59] 徳井翔梧, 吉田則裕, 崔恩瀞, 井上克郎, 局所性鋭敏型ハッシュを用いたコードクローン検出のためのパラメータ決定手法, in: 電子情報通信学会技術研究報告, Vol. 117, 2018, pp. 57–62.

[60] A. Andoni, P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, in: 2006 47th Annual IEEE Symposium on Foundations of Computer ScienceS, 2006, pp. 459–468.

[61] M. S. Charikar, Similarity estimation techniques from rounding algorithms, in: Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing, 2002, pp. 380–388.

[62] L. Jiang, G. Misherghi, Z. Su, S. Glondu, Deckard: Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, 2007, pp. 96–105.

[63] 肥後芳樹, 楠本真二, プログラム依存グラフを用いたコードクローン検出法の改善と評価, 情報処理学会論文誌 51 (12) (2010) 2149–2168.

[64] Y. Higo, S. Kusumoto, Enhancing quality of code clone detection with program dependency graph, in: Proceedings of the 16th Working Conference on Reverse Engineering, 2009, pp. 315–316.

[65] 堀田圭佑, 楊嘉晨, 肥後芳樹, 楠本真二, 粗粒度なコードクローン検出手法の精度に関する調査, 情報処理学会論文誌 56 (2) (2015) 580–592.

[66] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes, Sourcerercc: Scaling code clone detection to big-code, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 1157–1168.

[67] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Trans. Softw. Eng. 33 (9) (2007) 577–591.

[68] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, Method and implementation for investigating code clones in a software system, Information and Software Technology 49 (9) (2007) 985–998.

[69] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent dirichlet allocation, Journal of machine Learning research 3 (Jan) (2003) 993–1022.

[70] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, M. M. Mia, Towards a big data curated benchmark of inter-project code clones, in: Proceedings of the International Conference on Software Maintenance and Evolution, 2014, pp. 476–480.

[71] Z. Li, S. Lu, S. Myagmar, Y. Zhou, Cp-miner: finding copy-paste and related

bugs in large-scale software code, IEEE Trans. Softw. Eng. 32 (3) (2006) 176–192.

[72] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 40–56.

[73] C. K. Roy, J. R. Cordy, An empirical study of function clones in open source software, in: Proceedings of the 15th Working Conference on Reverse Engineering, 2008, pp. 81–90.

[74] C. K. Roy, J. R. Cordy, NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in: Proceedings of the 16th IEEE International Conference on Program Comprehension, 2008, pp. 172–181.

[75] B. van Bladel, S. Demeyer, A novel approach for detecting type-iv clones in test code, in: Proceedings of the 13th International Workshop on Software Clones, 2019, pp. 8–12.

[76] J. R. Cordy, C. K. Roy, Tuning research tools for scalability and performance: The nicad experience, Science of Computer Programming 79 (2014) 158–171.

[77] H. Kim, Y. Jung, S. Kim, K. Yi, Mecc: memory comparison-based clone detector, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 301–310.

[78] A. Marcus, J. I. Maletic, Identification of high-level concept clones in source code, in: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, 2001, pp. 107–114.

[79] 佐々木裕介, 山本哲男, 早瀬康裕, 井上克郎, 大規模ソフトウェアシステムを対象としたファイルクローンの検出, 電子情報通信学会論文誌 J94-D (8) (2011) 1423–1433.

[80] E. Duala-Ekoko, M. P. Robillard, Clone region descriptors: Representing and tracking duplication in source code, ACM Trans. Softw. Eng. Methodol. 20 (1) (2010) 3:1–3:31.

[81] R. Holmes, G. C. Murphy, Using structural context to recommend source code examples, in: Proceedings of the 27th International Conference on Software Engineering, 2005, pp. 117–125.

[82] I. Keivanloo, J. Rilling, Y. Zou, Spotting working code examples, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 664–675.

[83] K. Inoue, Y. Sasaki, P. Xia, Y. Manabe, Where does this code come from and where does it go? — integrated code history tracker for open source

systems, in: Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 331–341.

[84] G. Zhao, J. Huang, Deepsim: Deep learning code functional similarity, in: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 141–151.

[85] Z. Li, S. Lu, S. Myagmar, Y. Zhou, Cp-miner: finding copy-paste and related bugs in large-scale software code, IEEE Trans. Softw. Eng. 32 (3) (2006) 176–192.

[86] 横井一輝, 崔恩瀞, 吉田則裕, 井. 克郎, 情報検索技術に基づく細粒度ブロッククローン検出, コンピュータ ソフトウェア 35 (4) (2018) 16–36.

[87] 横井一輝, 崔恩瀞, 吉田則裕, 井. 克郎, コード片のベクトル表現に基づく大規模コードクローン集合の特徴調査, in: ソフトウェアエンジニアリングシンポジウム 2018 論文集, 2018, pp. 192–199.

[88] K. Yokoi, E. Choi, N. Yoshida, K. Inoue, Investigating vector-based detection of code clones using bigclonebench, in: Proceedings of the 25th Asia-Pacific Software Engineering Conference, 2018, pp. 699–700.

[89] Q. Le, T. Mikolov, Distributed representations of sentences and documents, in: Proceedings of the 31st International Conference on International Conference on Machine Learning, 2014, pp. 1188–1196.

[90] 藤原裕士, 崔恩瀞, 吉田則裕, 井上克郎, 順伝播型ニューラルネットワークを用いた類似コードブロック検索の試み, in: ソフトウェアエンジニアリングシンポジウム 2018 論文集, 2018, pp. 24–33.

[91] M. White, M. Tufano, C. Vendome, D. Poshyvanyk, Deep learning code fragments for code clone detection, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 87–98.

[92] W. Hua, Y. Sui, Y. Wan, G. Liu, G. Xu, Fcca: Hybrid code representation for functional clone detection using attention networks, IEEE Transactions on Reliability 70 (1) (2021) 304–318.

[93] G. E. Hinton, R. R. Salakhutdinov, Reducing the dimensionality of data with neural networks, Science 313 (5786) (2006) 504–507.

[94] W. Wang, G. Li, B. Ma, X. Xia, Z. Jin, Detecting code clones with graph neural network and flow-augmented abstract syntax tree, in: Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering, 2020, pp. 261–271.

[95] E. Enslen, E. Hill, L. Pollock, K. Vijay-Shanker, Mining source code to

automatically split identifiers for software analysis, in: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, 2009, pp. 71–80.

[96] R. Řehůřek, P. Sojka, Software Framework for Topic Modelling with Large Corpora, in: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, 2010, pp. 45–50.

[97] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, arXiv preprint arXiv:1301.3781.

[98] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, Proceedings of the 26th International Conference on Neural Information Processing Systems (2013) 3111–3119.

[99] E. Choi, N. Yoshida, T. Ishio, K. Inoue, T. Sano, Extracting code clones for refactoring using combinations of clone metrics, in: Proceedings of the 5th International Workshop on Software Clones, 2011, p. 7–13.

[100] M. Brand, Fast low-rank modifications of the thin singular value decomposition, Linear Algebra and its Applications 415 (1) (2006) 20–30.

[101] R. Socher, C. C.-Y. Lin, A. Y. Ng, C. D. Manning, Parsing natural scenes and natural language with recursive neural networks, in: Proceedings of the 28th International Conference on International Conference on Machine Learning, 2011, pp. 129–136.

[102] M. J. Kusner, Y. Sun, N. I. Kolkin, K. Q. Weinberger, From word embeddings to document distances, in: Proceedings of the 32nd International Conference on International Conference on Machine Learning, Vol. 37, 2015, pp. 957–966.

[103] 藤原裕士, 崔恩瀞, 吉田則裕, 井上克郎, 深層学習を用いたソースコード分類手法の比較調査, 電子情報通信学会論文誌 J104-D (8) (2021) 622–635.

[104] 藤原裕士, 森彰, 井上克郎, 回帰モデルを用いたコードクローン検出手法の提案と汎化性能の評価, 電子情報通信学会論文誌 J104-D (9) (2021) 678–689.

[105] C. Becker, F. Fagerholm, R. Mohanani, A. Chatzigeorgiou, Temporal discounting in technical debt: How do software practitioners discount the future?, in: Proceedings of the Second International Conference on Technical Debt, 2019, pp. 23–32.

[106] S. Murer, B. Bonati, Managed evolution: a strategy for very large information systems, Springer Science & Business Media, 2010.

[107] K. A. Nasr, H.-G. Gross, A. van Deursen, Realizing service migration in industry—lessons learned, Journal of Software: Evolution and Process 25 (6)

(2013) 639–661.

[108] Y. Cai, R. Kazman, C. V. Silva, L. Xiao, H.-M. Chen, Chapter 6 - a decision-support system approach to economics-driven modularity evaluation, in: Economics-Driven Software Architecture, Morgan Kaufmann, 2014, pp. 105–128.

[109] D. Cui, L. Fan, S. Chen, Y. Cai, Q. Zheng, Y. Liu, T. Liu, Towards characterizing bug fixes through dependency-level changes in apache java open source projects, Science China Information Sciences 65.

[110] R. Leitch, E. Stroulia, Assessing the maintainability benefits of design restructuring using dependency analysis, in: Proceedings of the 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry, 2003, p. 309.

[111] H. Rebêl, R. M. F. Lima, U. Kulesza, M. Ribeiro, Y. Cai, R. Coelho, C. Sant'Anna, A. Mota, Quantifying the effects of aspectual decompositions on design by contract modularization: a maintenance study, Int. J. Softw. Eng. Knowl. Eng. 23 (7) (2013) 913–942.

[112] L. Xiao, Y. Cai, R. Kazman, R. Mo, Q. Feng, Detecting the locations and predicting the maintenance costs of compound architectural debts, IEEE Trans. Softw. Eng. 48 (9) (2022) 3686–3715.

[113] J. Bisbal, D. Lawless, R. Richardson, D. O'Sullivan, B. Wu, J. B. Grimson, V. P. Wade, A survey of research into legacy system migration (2007).

[114] E. Evans, Domain-driven design: tackling complexity in the heart of software, Addison-Wesley Professional, 2004.

[115] 独立行政法人情報処理推進機構, ソフトウェア開発見積りガイドブック, SEC BOOKS, 2006.

[116] 独立行政法人情報処理推進機構, ソフトウェア開発分析データ集 2022 (2022).

[117] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, B. Steece, Software Cost Estimation with COCOMO II, Prentice Hall, 2000.

[118] B. W. Boehm, Software engineering economics, IEEE Trans. Softw. Eng. SE-10 (1) (1984) 4–21.

[119] 小林健一, 松尾明彦, 井上克郎, 早瀬康裕, 上村学, 吉野利明, 大規模ソフトウェア保守のための影響波及量尺度インパクトスケール, 情報処理学会論文誌 54 (2) (2013) 870–882.

[120] T. A. Corbi, Program understanding: Challenge for the 1990s, IBM Syst.J. 28 (2) (1989) 294–306.

[121] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha,

S. A. Spoon, A. Gujarathi, Regression test selection for java software, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2001, p. 312–326.

[122] C. L. McClure, The three Rs of software automation : re-engineering, repository, reusability, Prentice Hall, 1992.

[123] A. Ko, H. H. Aung, B. Myers, Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks, in: Proceedings of the 27th International Conference on Software Engineering, 2005, pp. 126–135.

[124] L. Briand, Y. Labiche, G. Soccar, Automating impact analysis and regression test selection based on uml designs, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 252–261.

[125] 早瀬康裕, 松下誠, 楠本真二, 井上克郎, 小林健一, 吉野利明, 影響波及解析を利用した保守作業の労力見積りに用いるメトリックスの提案, 電子情報通信学会論文誌 D J90-D (10) (2007) 2736–2745.

[126] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, J. M. Zhang, Large language models for software engineering: Survey and open problems, in: Proceedings of International Conference on Software Engineering: Future of Software Engineering, 2023, pp. 31–53.

[127] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, H. Chen, From llms to llm-based agents for software engineering: A survey of current, challenges and future, arXiv preprint arXiv:2408.02479.

[128] S. Dou, J. Shan, H. Jia, W. Deng, Z. Xi, W. He, Y. Wu, T. Gui, Y. Liu, X. Huang, Towards understanding the capability of large language models on code clone detection: a survey, arXiv preprint arXiv:2308.01191.

[129] A. T. V. Dau, H. T. Dao, A. T. Nguyen, H. T. Tran, P. X. Nguyen, N. D. Q. Bui, Xmainframe: A large language model for mainframe modernization, arXiv preprint arXiv:2408.04660.

[130] C. Diggs, M. Doyle, A. Madan, S. Scott, E. Escamilla, J. Zimmer, N. Nekoo, P. Ursino, M. Bartholf, Z. Robin, et al., Leveraging llms for legacy code modernization: Challenges and opportunities for llm-generated documentation, arXiv preprint arXiv:2411.14971.

[131] A. Ferri, T. Coggrave, S. Sheth, Legacy Modernization meets GenAI, `https://martinfowler.com/articles/legacy-modernization-gen-ai.html`.

# 付録

## 付録 1: ベクトル表現とハイパーパラメータ

| ベクトル表現 | ハイパーパラメータ |
|---|---|
| LSI | トピック数:200 |
| LDA | トピック数:100 |
| PV-DBoW | ベクトルサイズ:300，ウィンドウサイズ:15，エポック数:20 |
| PV-DM | ベクトルサイズ:300，ウィンドウサイズ:5，エポック数:20 |
| WV-avg | ベクトルサイズ:300，ウィンドウサイズ:5，エポック数:20 |

# 付録 2: 手法ごとのハイパーパラメータ設定

| 手法 | ハイパーパラメータ |
|---|---|
| LSI+NN | トピック数:200, レイヤーサイズ:200-100,<br>エポック数:4, 初期学習率:0.001,<br>L2 正則化の λ:0.00003 |
| LDA+NN | トピック数:100, レイヤーサイズ:100-100,<br>エポック数:4, 初期学習率:0.001,<br>L2 正則化の λ:0.00003 |
| PV-DBoW+NN | ベクトルサイズ:300, ウィンドウサイズ:15,<br>エポック数（PV-DBoW）:20,<br>レイヤーサイズ:300-100, エポック数:4,<br>初期学習率:0.001, L2 正則化の λ:0.00003 |
| PV-DM+NN | ベクトルサイズ:300, ウィンドウサイズ:5,<br>エポック数（PV-DM）:20,<br>レイヤーサイズ:300-100, エポック数:4,<br>初期学習率:0.001, L2 正則化の λ:0.00003 |
| WV-avg+NN | ベクトルサイズ:300, ウィンドウサイズ:5,<br>エポック数（Word2Vec）:20,<br>レイヤーサイズ:300-100, エポック数:4,<br>初期学習率:0.001, L2 正則化の λ:0.00003 |
| DeepSim | レイヤーサイズ:88-6, (128x6-256-64)-128-32,<br>エポック数:4, 初期学習率:0.001,<br>L2 正則化の λ:0.00003, ドロップアウト:0.75 |
| FA-AST+GMN | レイヤーサイズ:100, ベクトル次元数:100,<br>エポック数:4, 初期学習率:0.001,<br>バッチサイズ:32 |