The University of Osaka
Institutional Knowledge Archive

Title	ソースコードの類似性分析に基づくソフトウェア保守支援に関する研究
Author(s)	吉田, 則裕
Citation	大阪大学, 2009, 博士論文
Version Type	VoR
URL	https://hdl.handle.net/11094/2431
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

ソースコードの類似性分析に基づく
ソフトウェア保守支援に関する研究

提出先　　大阪大学大学院情報科学研究科

提出年月　2009年1月

吉田　則裕

内容梗概

長期にわたって運用される大規模ソフトウェアが増加したことから，保守作業の効率化が重要な課題として取り上げられるようになった．ソフトウェアに対する保守作業の効率を下げている要因の１つとして，ソースコード中の類似コード片が挙げられる．類似コード片とは，ソースコードの一部分（コード片）のうち，他のコード片と類似性が高いものを指す．ソースコード中のあるコード片を修正すると，その類似コード片を同時に修正する作業が必要になることが多い．例えば，ソースコード中に欠陥が見つかった場合には，その欠陥を含むコード片の類似コード片を探し，検査する必要がある．類似コード片に対する保守作業の効率を上げるためには，ソースコード上における類似コード片の位置情報を記録する方法や，集約可能な類似コード片の集合を抽出し，サブルーチンにする方法が考えられるが，大規模ソースコードに含まれる類似コード片に対し手作業でこれらを行うには大きな労力が必要となる．

これまでに，類似コード片の中でもコードクローン（トークン列や構文木が等価であるコード片が存在するコード片）に着目し，自動検出を行うツール（コードクローン検出ツール）が数多く開発されている．これらを用いることで，修正すべきコード片のコードクローンを検出し，同様の修正を加える作業を支援することができる．また，クローンセット（コードクローンの同値類）を１つのサブルーチンにする作業の容易さを表すメトリクスを提示するなど，自動検出したコードクローンの集約作業を支援する手法も提案されている．

しかし，既存のコードクローン検出ツールの問題点として，以下が挙げられる．

1. 同一クローンセットに属するコード片の集合を集約する際に，それらと呼び出し関係を持つメソッドを含むコード片が属するクローンセット，もしくはそれらと変数を共有するコード片が属するクローンセットについても集約する必要が生じることがある．しかし，このようなクローンセット間の依存関係（あるクローンセットを集約するためには，他のクローンセットの集約が必要という関係）を，コードクローン検出ツールは検出しない．

2. トークン列や構文木が等価であるコード片が存在するコード片のみを検出するため，構文上に差異がある類似コード片を検出できないことが多い．例えば，あるコード片を複製し，例外処理やログ出力文を追加すると，それらコード片はコードクローンとして判定されないことが多い．しかし，複製元

のコード片を修正すると，例外処理やログ出力文を追加した複製先のコード片についても，同時に修正する必要が生じることが多い．

本論文では，1. で挙げた問題点を解決するため，チェーンドクローンセット（同時に集約を検討すべきクローンセットの集合）を提示することで，リファクタリング支援を行う手法を提案する．本手法は，クローンセット中に存在するメソッド呼び出し関係，およびメソッドと変数の利用関係を解析することで，ソースコード中からチェーンドクローンセットを検出する．更に，検出したチェーンドクローンセットを含むクラスからなる集合の継承関係から，適用可能なリファクタリングパターン（メソッドの抽出や引き上げ，親クラスの抽出）を提示する．適用実験において，提案手法をオープンソースソフトウェアのソースコードに適用したところ，多くのチェーンドクローンセットを検出することができ，そのうちのいくつかは提案手法が提示するリファクタリングパターンを適用することができた．

また，2. で挙げた問題点を解決するために，識別子の類似性に基づく類似コード片検索手法を提案する．本手法は，クエリとしてコード片を与えると，識別子の類似性に基づいて対象ソースコードから類似関数（クエリとして与えられたコード片の類似コード片を含む関数）を検索する．具体的には，まず自然言語処理の分野で提案されている類義語特定法を用いて，語（識別子を分割・正規化した後の文字列）の類義語を特定する．次に，クエリとして与えられたコード片に含まれる全ての語について，同一もしくは類義語である語を含む関数を検出し，類似関数として提示する．適用実験として，本手法を用いて類似した欠陥を含むコード片の検索を行ったところ，類似した欠陥の多くを提示できることを確認した．また，本手法と既存ツール（grep やコードクローン検出ツール CCFinder）との比較実験を行い，それぞれの検索結果が持つ特徴を確認した．

論文一覧

主要論文

[1-1] Norihiro Yoshida, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: "On Refactoring Support Based on Code Clone Dependency Relation", Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005), pp.16:1-16:10, Como, Italy, September 2005 (国際会議録).

[1-2] 吉田 則裕, 肥後 芳樹, 神谷 年洋, 楠本 真二, 井上 克郎: "コードクローン間の依存関係に基づくリファクタリング支援", 情報処理学会論文誌, Vol.48, No.3, pp.1431-1442, 2007 年 3 月 (学術論文).

[1-3] 吉田 則裕, 服部 剛之, 早瀬 康裕, 井上 克郎: "類義語の特定に基づく類似コード片検索法", 情報処理学会論文誌 [条件付採録] (学術論文).

関連論文

[2-1] 肥後 芳樹, 吉田 則裕, 楠本 真二, 井上 克郎: "産学連携に基づいたコードクローン可視化手法の改良と実装", 情報処理学会論文誌, Vol.48, No.2, pp.811-822, 2007 年 2 月 (学術論文).

[2-2] Norihiro Yoshida, Katsuro Inoue: "Towards an Investigation of Opportunities for Refactoring to Design Patterns", Proceedings of the 1st International Workshop on Software Patterns and Quality (SPAQu'07), pp.61-62, Nagoya, Japan, December 2007 (国際会議録).

[2-3] Norihiro Yoshida, Takashi Ishio, Makoto Matsushita, Katsuro Inoue: "Retrieving Similar Code Fragments based on Identifier Similarity for Defect Detection", Proceedings of the 1st International Workshop on Defects in Large Software Systems (DEFECTS 2008), pp.41-42, Seattle, USA, July 2008 (国際会議).

[2-4] 森崎 修司, 吉田 則裕, 肥後 芳樹, 楠本 真二, 井上 克郎, 佐々木 健介, 村上 浩二, 松井 恭: “コードクローン検索による類似不具合検出の実証的評価”, 電子情報通信学会論文誌 D, Vol.J91-D, No.10, pp. 2466-2477, 2008 月 10 月 (学術論文).

[2-5] 馬場 慎太郎, 吉田 則裕, 楠本 真二, 井上 克郎: “Fault-Prone モジュール予測へのコードクローン情報の適用”, 電子情報通信学会論文誌 D, Vol.J91-D, No.10, pp.2559-2561, 2008 年 10 月 (学術論文).

謝辞

本研究の遂行するにあたり，常日頃より適切な御指導を賜わりました，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 井上 克郎 教授に，心から深く感謝申し上げます.

本論文を執筆するにあたり，適切な御助言と御指導を頂きました，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 増澤 利光 教授，楠本 真二 教授に心から感謝致します.

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在籍中に，適切な御助言と御指導を頂きました，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 萩原 兼一 教授，八木 康史 教授 に感謝致します.

本研究を遂行するにあたり，直接具体的な御指導と御助言を頂きました，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 松下 誠 准教授，石尾 隆 助教，肥後 芳樹 助教，早瀬 康裕 特任助教，ならびに産業総合研究所 神谷 年洋 氏に心より御礼申し上げます.

本研究に関して，貴重な御助言を頂きました，株式会社富士通研究所 松尾 昭彦 氏，小林 健一 氏，前田 芳晴 氏，ならびに株式会社富士通東北システムズ 須藤 茂雄 氏に厚く御礼申し上げます.

提案手法の実現と評価を行うにあたり，様々な御協力を頂いた，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 服部 剛之 氏（現 日立電子サービス株式会社）に深く感謝致します.

最後に，井上研究室の皆様の御助言, 御協力に御礼申し上げます.

目 次

第 1 章　はじめに **1**
1.1　ソフトウェア保守 1
1.2　ソースコードの類似性分析 4
1.2.1　類似コード片 4
1.2.2　コードクローン検出ツール CCFinder 5
1.2.3　その他のコードクローン検出ツール 9
1.2.4　類似コード片を対象としたリファクタリング 11
1.3　既存手法の問題点 13
1.4　本論文の概要 16

第 2 章　コードクローン間の依存関係に基づくリファクタリング支援 **19**
2.1　導入 19
2.2　リファクタリング支援環境 Aries 20
2.3　提案手法 21
2.3.1　本研究の動機 21
2.3.2　チェーンドクローン 23
2.3.3　チェーンドクローンセットに対するリファクタリング 25
2.3.4　チェーンドクローンセットの分類 27
2.3.5　チェーンドクローンセットの分類を目的としたメトリクス . 28
2.3.6　実装 29
2.4　適用実験 32
2.4.1　概要 32
2.4.2　チェーンドクローンセットの検出 33
2.4.3　チェーンドクローンセットに対するリファクタリングの例 . 34
2.4.4　考察 36
2.5　関連研究 38
2.6　結論 39

第 3 章　類義語の特定に基づく類似コード片検索法 **41**
3.1　導入 41
3.2　類似コード片検索 43
3.2.1　grep を用いた類似コード片検索 44

3.2.2 CCFinder を用いた類似コード片検索 45
3.3 提案手法 46
3.3.1 手順 1（語の抽出） 46
3.3.2 手順 2（類義語の特定） 47
3.3.3 手順 3（入力コード片との照合） 49
3.4 適用実験 50
3.4.1 提案手法の適用実験 51
3.4.2 grep や CCFinder との比較実験 56
3.5 考察 58
3.5.1 語のクラスタリングに用いる閾値について 58
3.5.2 語のクラスタリングについて 60
3.5.3 grep との比較について 60
3.5.4 CCFinder との比較について 62
3.6 関連研究 62
3.7 結論 63

第 4 章 むすび **65**
4.1 まとめ 65
4.2 今後の研究方針 66

図 目 次

1.1 類似コード片 5
1.2 クローンペアとクローンセット 6
1.3 コードクローン検出例 8
1.4 類似コード片を対象とした “Extract Method” 12
1.5 類似コード片を対象とした “Pull Up Method” 12
1.6 類似コード片を対象とした “Extract SuperClass” 13
1.7 類似コード片を対象とした “Form Template Method” 14
1.8 類似コード片を対象とした “Consolidate Duplicate Conditional Fragments” 15
1.9 クローンセット間に存在する依存関係 16
1.10 類似コード片間の差異 17

2.1 DCH メトリクスの算出例 21
2.2 クローンセット間に存在する依存関係 22
2.3 呼び出し関係と呼び出し関係を表すグラフ 23
2.4 フィールド変数の使用と共有関係を表すグラフ（括弧内の数字は対応する関係を表す） 24
2.5 ケース 1 25
2.6 ケース 2 とケース 3 26
2.7 ケース 4 27
2.8 提案するメトリクスの算出例 30
2.9 Chained Clone Set Selection View 31
2.10 Chained Clone Set View 32
2.11 Source Code View 33
2.12 ANTLR から検出されたチェーンドクローンセットの例 35
2.13 図 2.12 のチェーンドクローンセットをリファクタリングした例 . . 36
2.14 JBoss から検出されたチェーンドクローンセットの例 36
2.15 図 2.14 のチェーンドクローンセットをリファクタリングした例 . . 37
2.16 分類 4 のチェーンドクローンセットの例 37

3.1 類似コード片 43
3.2 類似コード片間の差異 44

3.3 図 3.2(a) のコード片に対する修正 44
3.4 grep を用いた類似コード片検索 45
3.5 提案手法の概要 46
3.6 語の出現回数を表す行列の例 47
3.7 類義語の特定に用いる行列 48
3.8 入力コード片と対象関数の照合 50
3.9 かんなの実験結果 53
3.10 SPARS-J の実験結果 54
3.11 コード片 CF_A，CF_B，CF_C 55

表 目 次

2.1 チェーンドクローンセットの分類 28
2.2 クローンセットの検出結果 40
2.3 チェーンドクローンセットの検出結果 40

3.1 適用対象のソフトウェア 51
3.2 各コード片の検索結果 56
3.3 各クラスタに属していた語（抜粋） 57
3.4 コード片に含まれる語に属するクラスタ 57
3.5 grep による検索の結果 59

第1章　はじめに

1.1　ソフトウェア保守

コンピュータシステムの用途が多様化した現代社会において，ソフトウェアが担う社会的役割は極めて大きいと言える．ソフトウェアには高い信頼性が求められる一方で，その開発にかけられる時間や投入できる人的，計算機資源は限られている．そこで，信頼性の高いソフトウェアを効率的に開発する方法の実現を目指した研究が盛んに行われている．このような研究分野は，ソフトウェア工学と呼ばれる．

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，“納入後，ソフトウェアに対して加えられる，フォールト修正，性能または他の性質改善，変更された環境に対するプロダクトの適応のための改訂” のことを指す [33]．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 [35]．

修正保守 発見された問題を修正するために，納入後に実施される，ソフトウェア・プロダクトの対処的改変．

適応保守 変化した，または変化しつつある環境において，ソフトウェア・プロダクトを続けて使用可能なように維持するために，納入後に実施される，ソフトウェア・プロダクトの改変．

完全化保守 性能または保守性を改善するため，納入後に実施される，ソフトウェア・プロダクトの改変．

予防保守 ソフトウェア・プロダクトのなかに潜む，潜在的なフォールトが，効果的なフォールトに転じる前に，それを検出し，修正するために，納入後に実施される，ソフトウェア・プロダクトの改変．

長期間にわたって運用されるソフトウェアの開発では，一般に保守作業にかかるコストが大きいことが知られている [65]．そのため，保守作業の支援を目指して，手法の研究やツールの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の分析を行うものが多い．保守作業を行う開発者は，ソースコードのどこをどのように変更するか，変更した後にどのようなテストを実施するかを決定するた

めに，ソースコードの分析を行う．しかし，ソースコードが大規模である場合，手動で分析を行うことは困難である．そこで，コンピュータを用いてソースコードを分析することで，保守作業に有用と考えられる情報を自動的に抽出する手法やツールが数多く提案されている．それら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 [17].

ソフトウェア開発環境の中には，Imagix 4D[34] 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[32] 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) [15] 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 [15]．設計の復元を行う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（頻繁に用いられる設計，文献 [25] 参照）の実装部分を自動的に特定するツール [59, 61] 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）を予め与えておくと，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，保守作業を行う上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 [55].

回帰テスト選択手法 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が変化する可能性があることが指摘されている [14, 56]．このように，変更が他の部分に影響することは波及効果 (Ripple Effect) [14] と呼ばれる．

波及効果が発生する可能性があるため，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 [56, 57].

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 [56]．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 [56]．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 が変化する可能性があるテストコードを提示する．

メトリクスの計測 ソースコードの保守性（保守しやすさ）の評価を行うメトリクスの代表的なものとして，CK メトリクス [16] 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 [64] を概ね満足していること [16]，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 [11] 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 [63]．例えば，メトリクス Tightness[63] は，C 言語における関数中の文のうち，全てのスライス[1]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 [53]．

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 [40]．リファクタリング [23, 54] とは，保守性の改善を目的とした変更作業のことである（詳細な定義は 1.2.4 節を参照）．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することができる．

[1]関数の全ての返値，および関数中で変更される全ての大域変数をスライス基準とするスライス

以上のように，保守作業を支援するために多くの研究がなされている．

大規模ソースコードの保守を行うにあたって，ソースコード中に類似部分が多く含まれることが問題にされており，類似性分析のための手法やツールが提案されている．次節以降，主にソースコードの類似性分析について述べるが，保守作業の現場において有効な支援を行うためには，類似性分析のみを単独で用いるのではなく，本節で述べたリバースエンジニアリングや回帰テスト選択手法，メトリクスの計測と組み合わせて使う必要がある．

1.2 ソースコードの類似性分析

1.2.1 類似コード片

ソフトウェアに対する保守作業の効率を下げている要因の 1 つとして，ソースコード中の**類似コード片**が指摘されている [7, 13, 37, 39, 42, 46, 47, 66]．類似コード片とは，ソースコード中の**コード片**（ソースコードの一部分）のうち，一致もしくは類似した要素（識別子や構文など）を含むコード片を持つものを指す．類似コード片は，以下の理由で作成される [13, 41]．

既存コードのコピーとペーストによる再利用

近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーとペーストによる場当たり的な既存コードの再利用が多く行われるようになった．

定型処理

定義上簡単で頻繁に用いられる処理．例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

プログラミング言語に適切な機能の欠如

抽象データ型や，ローカル変数を用いられない場合には，同じようなアルゴリズムを持った処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

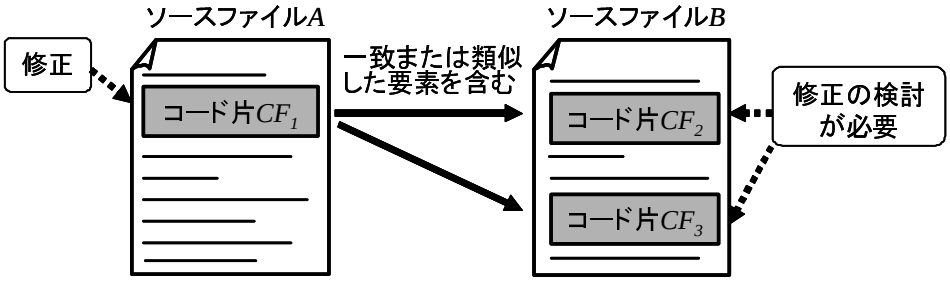


図 1.1: 類似コード片

特に，Linux や JDK (Java Development Kit) などの大規模ソースコードは大量の類似コード片を含むことが報告されている [39][47].

ソフトウェアの保守を行う際に，あるコード片を修正するとその全ての類似コード片を見つけ出し修正を行う必要が生じることがある．図 1.1 は，修正を要するコード片 CF_1 と，同様の修正を要する類似コード片 CF_2，CF_3 を表している．特に，ソースコード中に欠陥が見つかった場合には，その欠陥を含むコード片の類似コード片を探し，検査する必要がある [46][47][66]．しかし，ソフトウェア中の類似コード片を人手で探すためには大きな労力が必要となる．特に，大規模ソフトウェアが対象の場合，全ての類似コード片を人手で探すことはより困難となる．

類似コード片に対する保守作業の効率を上げる手段として，以下が考えられる．

(1) ソースコード上における類似コード片を検出し，その位置情報を記録する．

(2) 集約可能な類似コード片の集合を抽出し，単一のサブルーチンにする．

(1) で挙げた作業を支援するツールとして，コードクローン検出ツールが挙げられる．これらは，類似コード片の中でもコードクローン（トークン列や構文木が等価であるなど同値関係を持つコード片が存在するコード片）に着目し，自動検出を行うをツールである．1.2.2 節では，既存のコードクローン検出ツールについて述べる．

(2) で挙げた作業は，リファクタリング [23, 54]（ソフトウェアの外部から観測したときの動作を変化させることなく，ソースコードの品質を改善する作業）の一種と考えることができる．リファクタリングの解説書である文献 [23] において，開発者が類似コード片を集約する際の典型的な方法について述べられている．1.2.4 節では，この方法について述べる．

1.2.2 コードクローン検出ツール CCFinder

コードクローンの定義は研究者により様々であるが，類似コード片の中でも同値関係（例：トークン列や構文木が等しい関係など）を持つコード片が存在するも

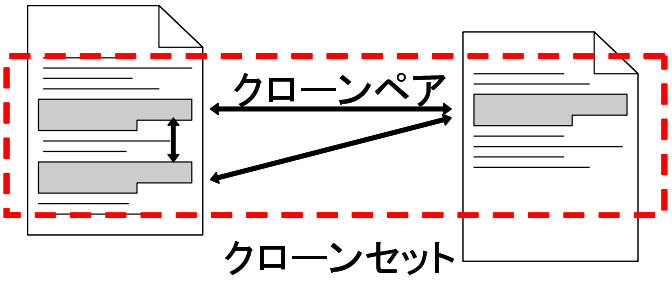


図 1.2: クローンペアとクローンセット

ののみを指すことがほとんどである [7, 13, 37, 39, 42]．本節では，コードクローン検出ツールの代表例として CCFinder[39] を紹介する．

まず，CCFinder おけるコードクローンの定義，および関連する語について述べる．あるトークン列中に存在する二つの部分トークン列 α，β が等価であるとき，α と β は互いにクローンであるという．またペア（α，β）をクローンペアと呼ぶ（図 1.2）．α，β それぞれを真に包含する如何なるトークン列も等価でないとき，α，β を極大クローンと呼ぶ．また，互いにクローンであるトークン列を同値としたときの同値類をクローンセットと呼ぶ（図 1.2）．ソースコード中でのクローンを特にコードクローンという [67]．

CCFinder は，単一または複数のファイルのソースコード中から全ての極大クローンを検出し，それをクローンペアの位置情報として出力する．CCFinder の持つ主な特徴は次の通りである．

細粒度のコードクローンを検出

字句解析を行うことにより，トークン単位でのコードクローンを検出する．

大規模ソフトウェアを実用的な時間とメモリで解析可能

例えば 10MLOC のソースコードを 68 分 (実行環境 Pentium3 650MHz RAM 1GB) で解析可能である [67]．

様々にプログラミング言語に対応可能

言語依存部分を取り替えることで，様々なプログラミング言語に対応できる．現在は，C/C++，Java，COBOL/COBOLS，Fortran，Emacs Lisp に対応している．またプレーンテキストに対しても，分かち書きされた文章として解析可能となっており，未対応の言語に対しても完全一致判定によるコードクローンはとることができる．

実用的に意味を持たないコードクローンを取り除く

- コードクローンは小さくなればなるほど偶然の一致の可能性が高くなるが，最小一致トークン数を指定することができるため，そのようなコードクローンを検出しないようにできる．

- モジュールの区切りを認識することで，複数のモジュールをまたがっているコードクローンを取り除いている．

ある程度の違いは吸収可能

- ソースコード中に含まれるユーザ定義名，定数をパラメータ化することで，その違いを吸収できる．
- クラススコープや名前空間による複雑な名前の正規化を行うことで，その違いを吸収できる．
- その他，テーブル初期化コード，可視性キーワード (protected, public, private 等)，コンパウンド・ブロックの中括弧表記等の違いも吸収することができる．

CCFinder のコードクローン検出手順（ソースコードを読み込んで，クローンペア情報を出力する）は大きく四つの過程から成り立っている [39].

ステップ 1（字句解析） ソースファイルを字句解析することによりトークン列に変換する．入力ファイルが複数の場合には，個々のファイルから得られたトークン列を連結し，単一のトークン列を生成する．

ステップ 2（変換処理） 実用上意味を持たないコードクローンを取り除くこと，及び，些細な表現上の違いを吸収することを目的とした変換ルールによりトークン列を変換する．例えば，この変換により変数名は同一のトークンに置換されるので，変数名が付け替えられたコード片もコードクローンであると判定することができる．

ステップ 3（検出処理） トークン列の中から指定された長さ以上一致している部分をクローンペアとして全て検出する．

ステップ 4（出力整形処理） 検出されたクローンペアについて，元のソースコード上での位置情報を出力する．

実際に，CCFinder によってどのようなコードクローンが検出されるのか例を示す．

図 1.3 に説明のための Java のソースコードを示す．このソースコードには，互いに似通った二つのメソッドが含まれ，左端には行番号が付されている．ここで，最小一致トークン数を 5 トークンに定め，図 1.3 のソースコードに対しコードクローン検出を行うと，図 1.3 中の `A1` (4 行目-6 行目) と `A2` (16 行目-17 行目), `B1` (8 行目- 10 行目) と `B2` (20 行目-22 行目), そして `C1` (12 行目) と `C2` (25 行目) がそれぞれクローンペアとして検出される．それぞれのクローンペアの長さは順に 7, 18, 6 トークンとなっている．見ての通り，A1 と A2 の間，B1 と B2 の間には次のような幾らかの違いが含まれているがコードクローンとして検出可能となっている．

- 名前空間の違い (e.g. “`org.apache.regexp.RE`” と “`RE`”).

```
    1. static void foo() throws RESyntaxException
    2. {
    3.    String a[] = new String [] {"123,400", "abc"};
A1  4.    org.apache.regexp.RE pat =
A1  5.              new org.apache.regexp.RE("[0-9,]+");
A1  6.    int sum = 0;
    7.    for (int i = 0; i < a.length; ++i)
B1  8.    {
B1  9.       if (pat.match(a[i])){
B1 10.          sum += Sample.parseNumber(pat.getParen(0));}
   11.    }
C1 12.    System.out.println("sum = " + sum);
   13. }
   14. static void goo(String [] a) throws RESyntaxException
   15. {
A2 16.    RE  exp = new RE("[0-9,]+");
A2 17.    int sum = 0;
   18.    int i   = 0;
   19.    while (i < a.length)
B2 20.    {
B2 21.       if (exp.match(a[i]))
B2 22.          sum += parseNumber(exp.getParen(0));
   23.       i++;
   24.    }
C2 25.    System.out.println("sum = " + sum);
   26. }
      :
      :
```

図 1.3: コードクローン検出例

- 変数名の違い (e.g. “pat” と “exp”).
- 改行とインデントの違い
- 中括弧表記の違い

これらの違いは，CCFinder のトークン変換処理によって吸収される．

1.2.3 その他のコードクローン検出ツール

以下に，その他のコードクローン検出ツールの中で主要なものを挙げる．

Covet

文献 [51] で定義された種々の特徴メトリクスの幾つかのメトリクス値を比較することによって，コードクローン検出を行う．現在，試作段階にあり，検出対象言語は，Java である．

CloneDR[13]

抽象構文木 (AST) の節点を比較することによって，コードクローン (類似部分木) の検出を行う．また，部分的に異なっているコードクローンも検出することが可能であり，検出したコードクローンを自動的に等価なサブルーチンやマクロに置き換えることも可能である．検出対象言語は，C/C++，COBOL，Java である．

Dup[4][5][6][7]

ユーザ定義名のパラメータ化を行った後，行単位の比較によりコードクローンを検出する．マッチングアルゴリズムには，サフィックス木探索 [28] を用いているため線形時間で解析可能である．

Duploc[20]

前処理として，空白やコメント等を取り除いた後，行単位 (のハッシュ値) での表検索を用いた比較によってコードクローンを検出する．また，コードクローンの散布図等の GUI を備えたツールであり，ソースコード参照支援を行う．検出対象言語は，C，COBOL，Python，Smalltalk である．

JPlag[45]

ソースコードを字句解析し，トークン単位での比較を行う．プログラム盗用の検出を目的として開発され，プログラム間の類似率を検出する．検出対象言語は，C/C++，Java である．

Komondoor らの手法 [42]

関数等にまとめるのに適したコードクローンの抽出を目的として，プログラム依存グラフ (PDG) 上での各節点の比較を行うことでコードクローン (同型 (isomorphic) 部分グラフ) を検出する．文字列比較や抽象構文木等を用いた方法では検出できなかった非連続コードクローンや，対応行の順番が異なるクローン，互いに絡みあったクローン等を検出可能である．[42] で作成されたツールの検出対象言語は，C である．

Krinke の手法 [43]

AST や Traditional PDG に似た Fine-grained PDG というグラフ上での類似

(similar) 部分グラフ (同型部分グラフではない) を検出することで，コードクローンが存在すると思しき場所を検出する．試作ツールの検出対象言語は，C である．

SMC[**10**][**8**][**9**]
まず特徴メトリクスによってコードクローンと思しきメソッドに絞り込む．次に絞り込まれたメソッドのペアに対し，表検索を用いることでメソッド単位のコードクローンを検出する．特徴メトリクスによって絞り込まれているため，実用上ほぼ線形時間で解析可能である．また検出されたペアのメソッドは，特徴により 18 種類に分類される．さらにそれぞれの分類については共通メソッドへの書き換え指針が示されている．

MOSS[**2**]
検出アルゴリズムは公開されていない．JPlag 同様，プログラム盗用の検出を目的として開発された．検出対象言語は，Ada，C/C++，Java，Lisp，ML，Pascal，Scheme である．

CP-Miner[**47**]
CP-Miner は，まずソースコード中のユーザー定義名を正規化（特殊文字に置換）した後，ソースコード中の各文をハッシュ値に変換する．つまり，ユーザ定義名が異なる文であってもその他が一致していれば，同じハッシュ値に変換する．続いて，ハッシュ値のシーケンスに対して，CloSpan アルゴリズム（Sequential Pattern Mining を行うアルゴリズムの一種）を適用することで，頻出するハッシュ値のシーケンスを求める．最後に，頻出するハッシュ値のシーケンスに対応するソースコードをコードクローンとして提示する．CloSpan アルゴリズムは，連続でない（不一致部分を含む）シーケンスであっても検出することができるため，それを用いている CP-Miner は不一致部分を含むコードクローンであっても検出できる．

Wahler らの手法 [**62**]
Wahler らの手法は，ソースコードの抽象構文木を表す XML 表現からコードクローンを検出する．具体的には，まず XML で記述された抽象構文木に対して，Frequent Itemset Mining アルゴリズムを適用することで，頻出する文の集合を特定する．そして，それら文をコードクローンとして提示する．この手法の特徴は，ソースコードを XML 形式に変換できれば，どのようなプログラミング言語のソースコードからでもコードクローンを検出できることである．いくつかのプログラミング言語はソースコードを XML 形式に変換する方法 [22, 50, 58] が提案されているため，Wahler らの手法を用いることで，それらプログラミング言語のソースコードからコードクローンを検出することができる．

Clone Miner[12]

Clone Minerは，字句解析後のトークン列に対してSuffix Arrayアルゴリズム[1]を適用することでコードクローンを検出する．CCFinderと同様に，コードクローン検出前にユーザ定義名を正規化（特殊文字に置換）する．加えて，Clone Minerはコードクローン検出後にFrequent Closed Itemset Mining[27]を適用することで，同一ファイルに含まれやすいクローンセットの集合を特定している．

1.2.4 類似コード片を対象としたリファクタリング

リファクタリングとは“外部からみたときの振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を変化させること”であると定義されている[23, 54].

Fowlerは文献[23]の中で，リファクタリングを検討すべき箇所にあらわれる特徴をBad Smellと呼び，その代表例として類似コード片を挙げている．類似コード片を取り除く手法として，次のような対処方法がある(以下，オブジェクト指向プログラミング言語，特にJavaを例にとって述べる).

“Extract Method”

ひとまとめにできるコード片がある場合に，新たなメソッドとして定義し，抽出されたコードを抽出先のメソッドへの呼び出し文に置き換える．特に重複したコードに限ったリファクタリングではないが，重複したコードで最も単純な例は，同一クラス内の複数メソッドに同じ式があるものである(図1.4参照).

“Pull Up Method”

同じ結果をもたらすメソッドが複数のサブクラスに存在した場合，それらを親クラスに引き上げる．最も単純なケースは，複数のメソッド本体が全く同じである場合である(図1.5参照)．重複したコードが兄弟クラスに存在した場合には，“Extract Method”を行ってから，“Pull Up Method”を行えばよい.

“Extract Class”

二つのクラスでなされるべき作業を一つのクラスで行っている際に，新たにクラスを作って，適当なフィールドとメソッドを元のクラスからそこに移動する．これも特に重複したコードに限ったリファクタリングではないが，全く関係のない複数のクラス間で，重複したコードが見られるときには，メソッド引き上げの代わりに別のクラスとして定義する.

“Extract SuperClass”

似通った特性を持つ複数のクラスがある場合に，新たに親クラスを作成して，

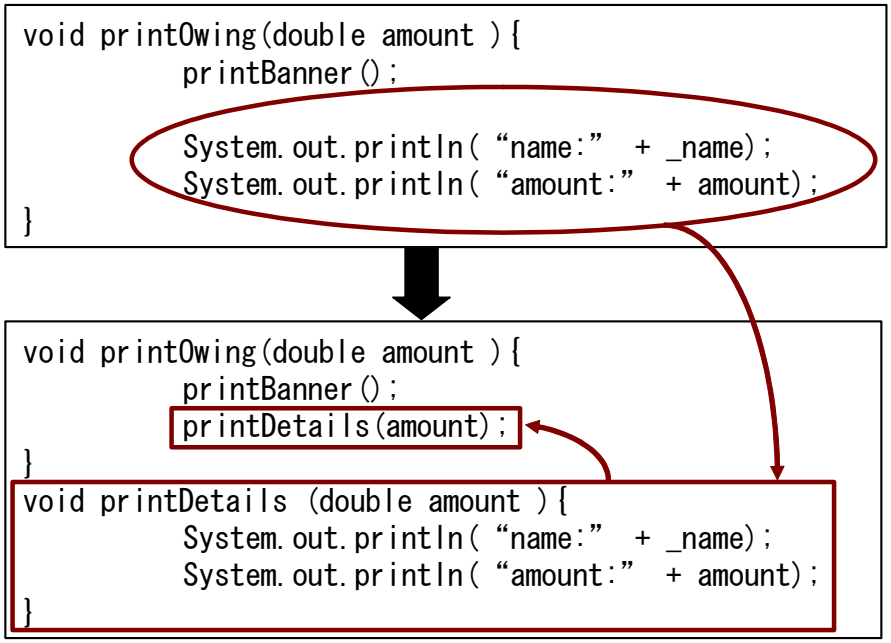


図 1.4: 類似コード片を対象とした “Extract Method”

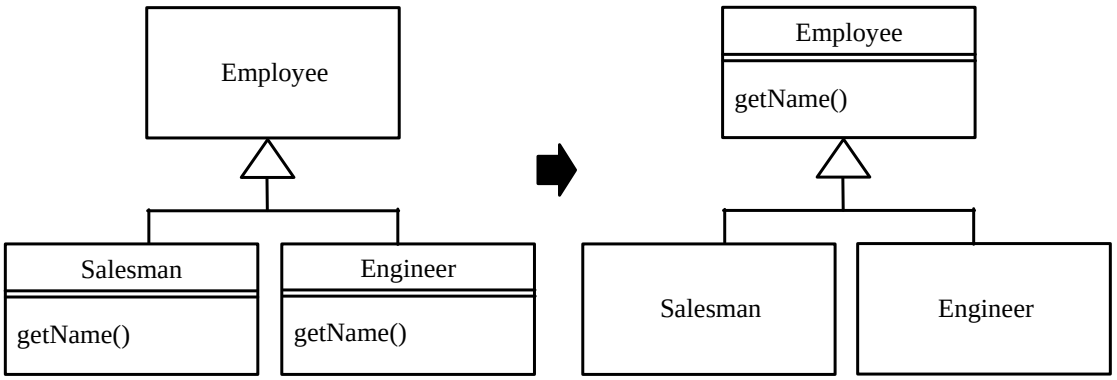


図 1.5: 類似コード片を対象とした “Pull Up Method”

共通の特性を移動する．“Extract Class” との違いは，継承するか委譲するかの違いである (図 1.6 参照).

“Form Template Method”

類似の処理を同じ順序で実行しているが，各処理が異なる場合，各処理を同じシグニチャを持つメソッドとして，親クラスに引き上げる．例えば，よく似た処理を同じ順番で実行しているものの，処理内容が違う場合には，順序の制御を親クラスに移動し，異なる処理については，元のクラスで行わせる (図 1.7 参照).

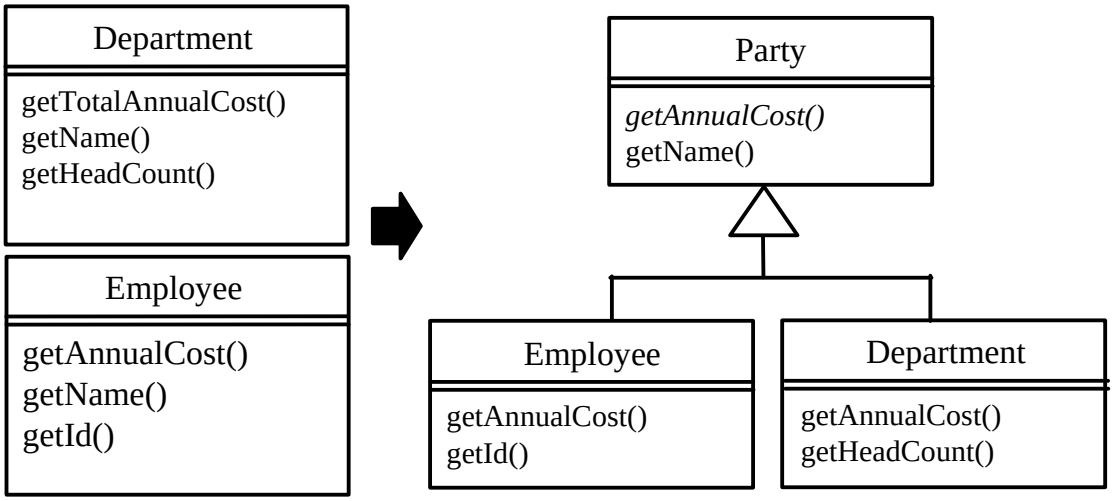


図 1.6: 類似コード片を対象とした “Extract SuperClass”

“Consolidate Duplicate Conditional Fragments”

条件式の全ての分岐に同じコード片がある場合，式の外側に移動する (図 1.8 参照)．条件記述以外にも，例外記述にも適用可能である．例えば，try ブロック内の例外の原因となる文の後，および全ての catch ブロックの中に重複コードがあるときは，`finally` ブロックに移動する．

“Replace Conditional with Polymorphism”

switch 文などは重複したコードを生成しやすくしている．同じような switch 文がある場合は，新たな分岐を追加した際に全ての switch 文を探して似たような変更をしなければならない場合も多く，新たな分岐での処理も他の分岐と比較し類似した処理が並ぶことが多いためである．その中でも特にオブジェクトの種類で分岐していた際には，ポリモーフィズムを利用した “Extract Method” 等で対処することができる．

1.3 既存手法の問題点

既存のコードクローン検出ツールの問題点として，以下が挙げられる．

- クローンセット間の依存関係を検出しない．

  同一クローンセットに属するコード片の集合を集約する際に，それらと呼び出し関係を持つメソッドを含むコード片が属するクローンセット，もしくはそれらと変数を共有するコード片が属するクローンセットについても集約する必要が生じることがある．しかし，このような**クローンセット間の依存関**

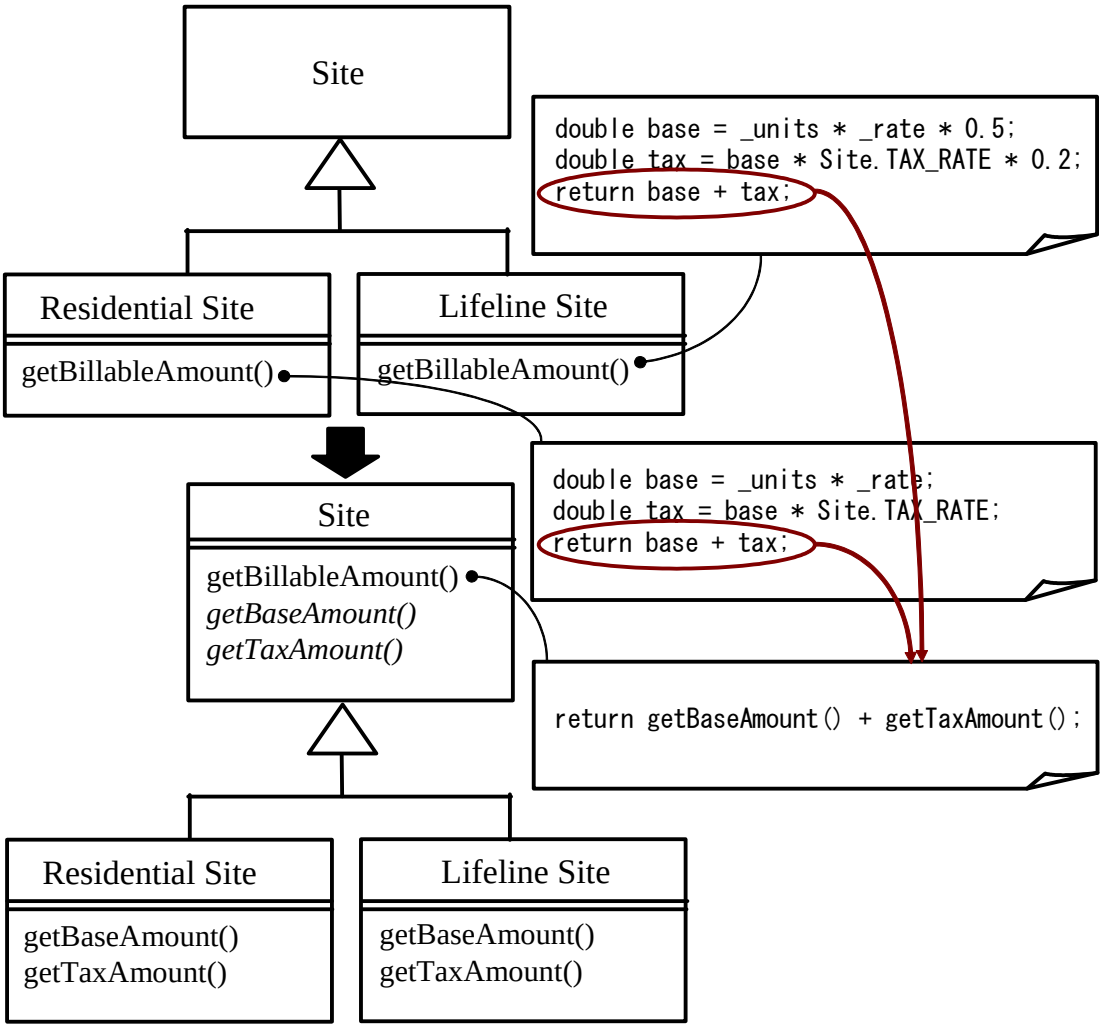


図 1.7: 類似コード片を対象とした “Form Template Method”

係（あるクローンセットを集約するためには，他のクローンセットの集約が必要という関係）を，コードクローン検出ツールは検出しない．

図 1.9(a) は，クローンセット間に存在する依存関係を表している．図 1.9(a) 中のクラス A，B にまたがる 3 つのクローンセットの間には，メソッド呼び出し関係やフィールドの利用関係が存在する．これらクローンセットのうち，メソッド a_1 とメソッド b_1 が属するクローンセットのみに対して集約を試みると，親クラスに新たに作成したメソッド s_1 から子クラスのメソッド a_2 と b_2 を呼び出すことが出来なくなる（図 1.9(b)）．ここで，メソッド a_2 とメソッド b_2 が属するクローンセット，加えて，それらメソッドがそれぞれ呼び出しているメソッド a_3 とメソッド b_3 が属するクローンセットについても同様に集約を行えば，メソッド a_1 とメソッド b_1 が属するクローンセットについても容易に集約することができる（図 1.9(c)）．

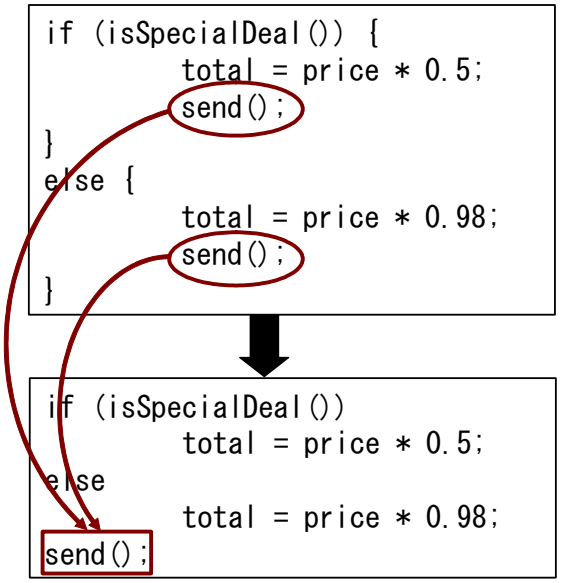


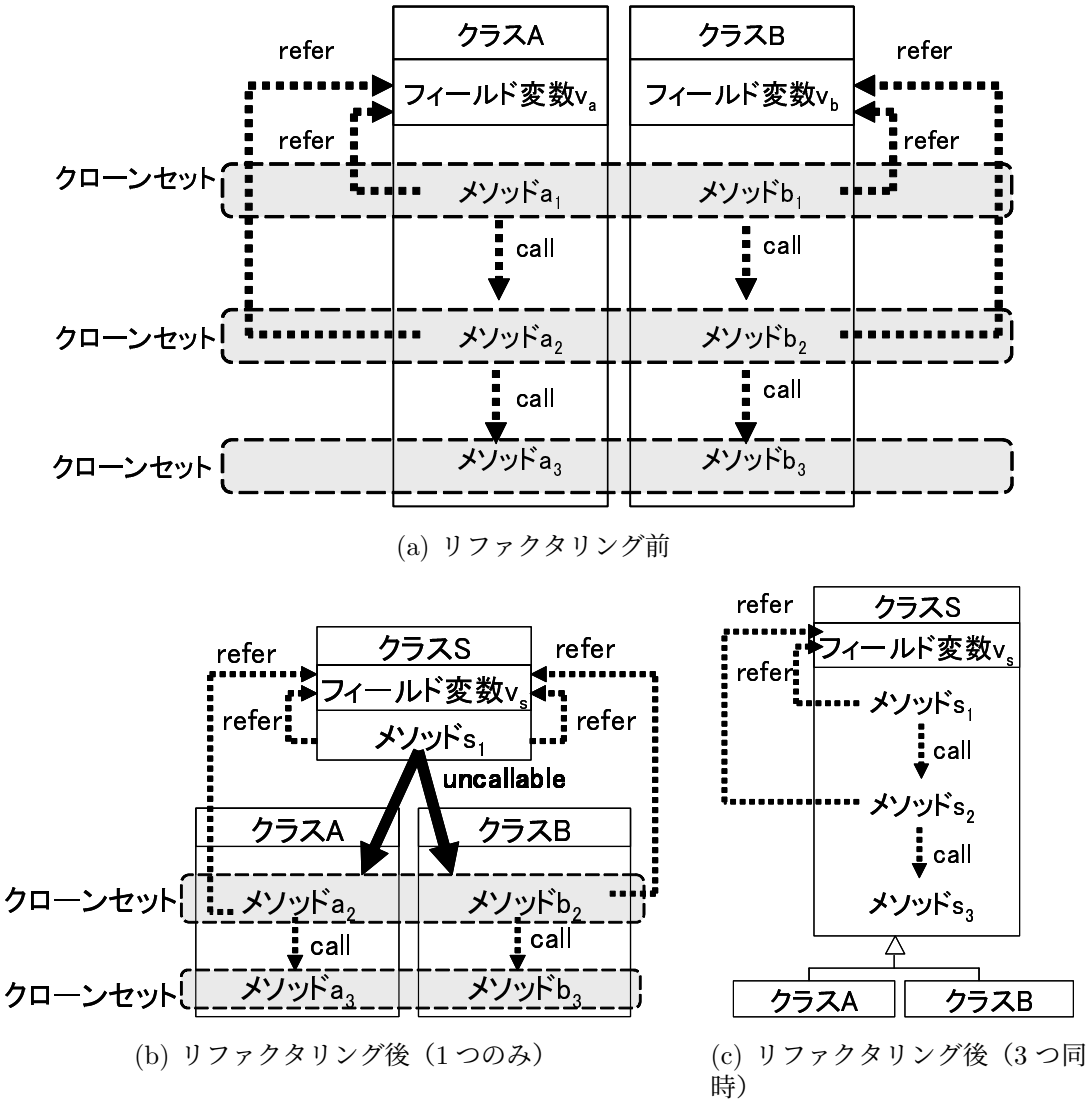
図 1.8: 類似コード片を対象とした “Consolidate Duplicate Conditional Fragments”

図 1.9(a) のように，あるクローンセットを集約するためには，他のクローンセットについても集約を行う必要が生じることがある．しかし，既存のコードクローン検出ツールは，クローンセット間に存在するメソッド呼び出し関係やフィールドの利用関係を検出しないため，開発者はクローンセット間の依存関係を知ることが困難である．

- トークンや構文上差異がある類似コード片を検出できないことが多い．

  あるコード片の類似コード片を検索する際に，コードクローン検出ツールを用いる方法が考えられる．コードクローン検出ツールを用いることで，あるコード片と（トークン列や構文木が等価であるなど）同値関係を持つコード片を検出することができる [30]．例えば，CCFinder[39] を利用すれば，あるコード片とトークン列が等価なコード片を列挙することができる．しかし，コードクローン検出ツールは同値関係を持つコード片のみを検出するため，トークン列や構文上差異がある類似コード片を検出できないことが多い．

  図 1.10 は，コードクローン検出ツールを用いても，トークン列上の些細な差異が原因で，類似コード片を検出できない例である．これら 2 つのコード片は，日本語入力システム “かんな” [71] バージョン 3.6 のソースコードに含まれていた類似コード片であり，各コード片の 3〜5 行目は，共にバッファからの読み込み処理を表している．図 1.10(a) の最終行の右辺は (short)S2TOS(buf) であるが，図 1.10(b) の最終行の右辺は (Ushort *)buf であり，トークン列上に差異が存在する．そのため，CCFinder を用いて，一方のコード片とトークン列が等価なコード片を検出したとしても，もう一方のコード片を検出する



(a) リファクタリング前

(b) リファクタリング後（1 つのみ）

(c) リファクタリング後（3 つ同時）

図 1.9: クローンセット間に存在する依存関係

ことはできない．

1.4 本論文の概要

本論文では，前節で挙げたコードクローン検出ツールの問題点を解決するために，以下の 2 つの手法を提案し，評価実験を行った結果について述べる．

1. コードクローン間の依存関係に基づくリファクタリング支援手法

   チェーンドクローンセット（同時に集約を検討すべきクローンセットの集合）を提示することで，リファクタリング支援を行う手法を提案する．本手法は，

```
ir_debug( Dmsg(10, "ProcWideReq7 start!!¥n") );

buf += HEADER_SIZE; Request.type7.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.number = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.yomilen =(short)S2TOS(buf);
```

バッファからの読み込み処理

(a) 欠陥を含むコード片

```
ir_debug( Dmsg(10, "ProcWideReq14 start!!¥n") );

buf += HEADER_SIZE; Request.type14.mode = L4TOL(buf);
buf += SIZEOFINT;   Request.type14.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type14.yomi = (Ushort *)buf;
```

バッファからの読み込み処理

(b) 同様の欠陥を含むコード片

図 1.10: 類似コード片間の差異

クローンセット中に存在するメソッド呼び出し関係，およびメソッドと変数の利用関係を解析することで，ソースコード中からチェーンドクローンセットを検出する．更に，検出したチェーンドクローンセットを含むクラスからなる集合の継承関係から，適用可能なリファクタリングパターン（“Extract Method” や，“Pull Up Method”，“Extract Superclass”）を提示する．適用実験では，オープンソースソフトウェアのソースコードに含まれているチェーンドクローンセットの規模を調査した．その後，検出したそれらチェーンドクローンセットに対して，提案手法が提示するリファクタリングパターンを適用した．

2. 識別子の類似性に基づく類似コード片検索手法

   本手法は，クエリとしてコード片を与えると，識別子の類似性に基づいて対象ソースコードから類似関数（クエリとして与えられたコード片の類似コード片を含む関数）を検索する．具体的には，まず自然言語処理の分野で提案されている類義語特定法を用いて，語（識別子を分割・正規化した後の文字列）の類義語を特定する．次に，クエリとして与えられたコード片に含まれる全ての語について，同一もしくは類義語である語を含む関数を検出し，類似関数として提示する．適用実験では，本手法を用いて類似した欠陥を含むコード片の検索を行い，類似した欠陥の多くを提示できるかを確認した．また，本手法と既存ツール（grep やコードクローン検出ツール CCFinder）との比較実験を行った．

第2章　コードクローン間の依存関係に基づくリファクタリング支援

2.1　導入

ソフトウェア保守性を改善する技術の1つとして，リファクタリング [23, 54] がある．Fowler は，リファクタリングを検討すべき箇所にあらわれる特徴を Bad Smell と呼び，その代表例として類似コード片（Duplicated Code）を挙げている [23]．また，類似コード片を単一のモジュールに集約する手法として，“Pull Up Method” や “Extract Method”，“Extract SuperClass” などのリファクタリングパターンを紹介している．

これまでに，著者が所属する研究グループでは，類似コード片の中でもコードクローンに着目し，コードクローン検出ツール CCFinder[39] およびリファクタリング支援環境 Aries[29] を開発している．CCFinder は，ソースコードに字句解析と正規化処理を行うことで得られたトークン列の同値性に基づいてコードクローン検出を行う．CCFinder の特徴は，表現上の差異があるコードクローンを検出できること，および百万行単位のソースコードであっても実用時間で解析できることである．Aries は，CCFinder の出力情報を基に，リファクタリングに適した単位（e.g. クラス，メソッド単位）でクローンセット（互いに一致または類似したコード片の集合）を検出し，更にメトリクスで特徴付けすることでリファクタリングパターンの提示を行う．

これまでに，Aries を用いて様々なソースコードを解析した結果，異なるクローンセットに含まれるコード片間に依存関係が存在する場合が確認されている．例えば，クローンセット S_a に2つのメソッド m_{a1}，m_{a2} が含まれ, 同様にクローンセット S_b に2つのコード片 m_{b1}，m_{b2} が含まれるときに，メソッド m_{a1} がメソッド m_{b1} を呼び出し，メソッド m_{a2} がメソッド m_{b2} を呼び出しているという場合である．

Aries は，上述した呼び出し関係の解析は行っていないため，ユーザは自らクローンセット S_a と S_b 間の呼び出し関係を把握する必要がある．もし，ユーザが S_a に対して m_{a1} と m_{b1}，m_{a2} と m_{b2} の呼び出し関係を考慮せずに集約を試みると，呼び出し関係が保存されない可能性がある．有効なリファクタリング支援を行うた

めには，クローンセット S_a と S_b は，まとめてユーザに提示するべきであると考えられる．

本章では，クローンセット間の依存関係を利用したリファクタリング支援手法を提案する．まず，異なるクローンセットに含まれるコード片間の依存関係に着目し，そのような依存関係を持つコード片の集合をチェーンドクローンセットと定義する．そして，チェーンドクローンセットの特徴に応じて，適用可能なリファクタリングパターンを提示するためのメトリクスを定義する．最後に，提案手法をリファクタリング支援ツールとして実装し，2つのオープンソースソフトウェアに適用することで有効性の評価を行う．

2.2 リファクタリング支援環境 Aries

リファクタリング支援環境 Aries は，CCFinder の出力情報を基に，リファクタリングに適した単位（e.g. クラス，メソッド単位）でクローンセットを検出し，更にメトリクスで特徴付けすることでリファクタリングパターンの提示を行う．

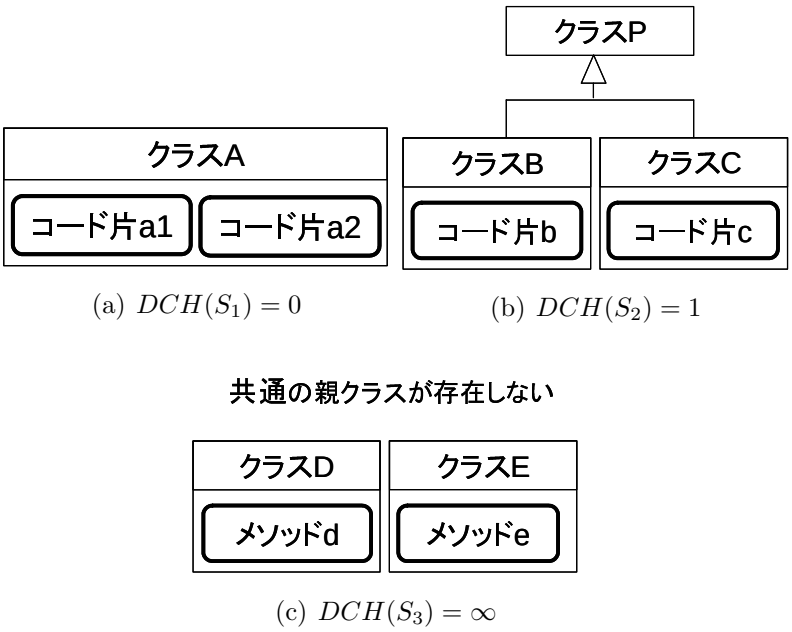
リファクタリングに適した単位とは，ソースコード上の構造的なまとまりのことである．クローンセットに含まれるコード片が構造的なまとまりを持っているなら，容易に集約することが出来る．現在，Aries は Java 言語を対象として実装されているため，用いる構造的なまとまりは以下の 12 種類である．

宣言	:	class { }, interface { }
メソッド	:	メソッド本体, コンストラクタ, スタティックイニシャライザ
文	:	if, for, while, do, switch, try, synchronized

クローンセットの特徴付けに用いるメトリクスの 1 つとして，分散度メトリクス **DCH(S)**[29] について説明する．クローンセット S はコード片 $f_1,\ f_2,\ \cdots,\ f_n$ を含んでいるとする．クラス C_i はコード片 f_i を含んでいるクラスとする．もしクラス $C_1,\ C_2,\ \cdots,\ C_n$ が共通の親クラスを持つ場合は，その共通の親クラスの中で，クラス階層的に最も下位（最も深い階層）に位置するクラスを C_p で表すとする．また $D(C_k,\ C_h)$ はクラス C_k と C_h のクラス階層における距離を表すとする．この時，

$$DCH(S) = max\ \{D(C_1,\ C_p),\ \cdots,\ D(C_n,\ C_p)\}$$

と表される．直観的には，$DCH(S)$ メトリクスはクローンセット S に含まれる各コード片間のクラス階層内における最大の距離を示す．図 2.1(a)〜(c) は，それぞれクローンセットに含まれる 2 つのコード片に対して，$DCH(S)$ メトリクスを算出した例である．$DCH(S)$ の値は，全てのコード片が 1 つのクラス内に存在する場合は 0 (図 2.1(a))，あるクラスとその直接の子クラス内に存在する場合は 1 とな



(a) $DCH(S_1) = 0$ (b) $DCH(S_2) = 1$

(c) $DCH(S_3) = \infty$

図 2.1: DCH メトリクスの算出例

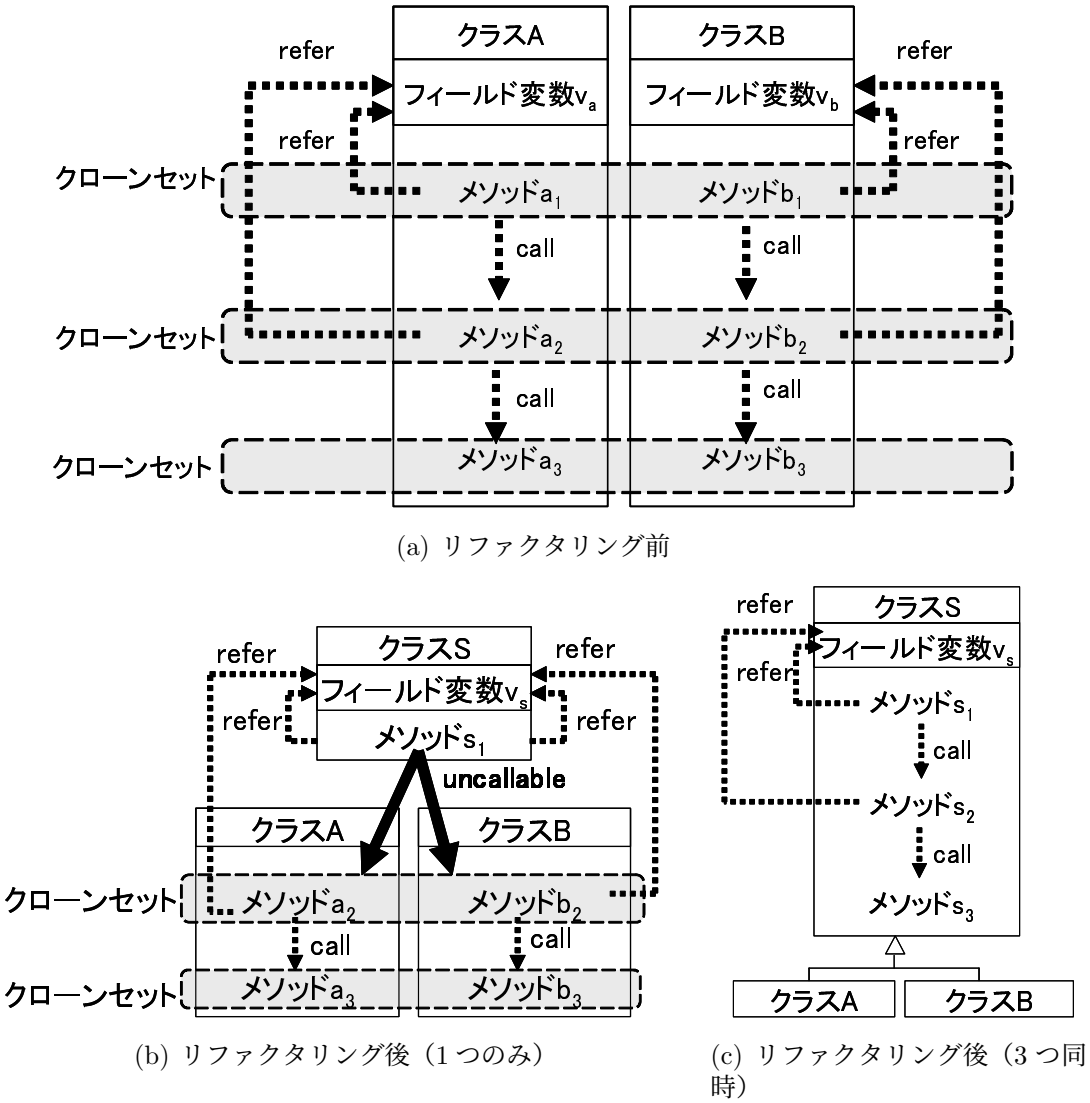
る (図 2.1(b))．例外的に，コードクローンが存在するクラスが共通の親クラスを持たない場合は ∞ とする (図 2.1(c))．このメトリクスは，クラスライブラリ等の修正不可能なクラスを除外したクラスを対象として計算される．

$DCH(S)$ メトリクスにより，クローンセット S のコード片を集約したモジュールを置くことが出来るクラス（集約先）を特定することが出来る．例えば，$DCH(S)$ メトリクスの値が 1 の場合は，そのクローンセットが存在するクラスの親クラスに集約できることがわかる．また，$DCH(S)$ メトリクスの値が ∞ の場合は，分析対象内に集約先位置になるクラスが存在しないため，クラスの作成，もしくは継承関係のないクラスへの集約を検討するべきであることがわかる．

2.3 提案手法

2.3.1 本研究の動機

同一クローンセットに属するコード片の集合を集約する際に，それらと呼び出し関係を持つメソッドを含むコード片が属するクローンセット，もしくはそれらと変数を共有するコード片が属するクローンセットについても集約する必要が生じることがある．しかし，このような**クローンセット間の依存関係**（あるクローンセットを集約するためには，他のクローンセットの集約が必要という関係）を，コードクローン検出ツールは検出しない．



(a) リファクタリング前

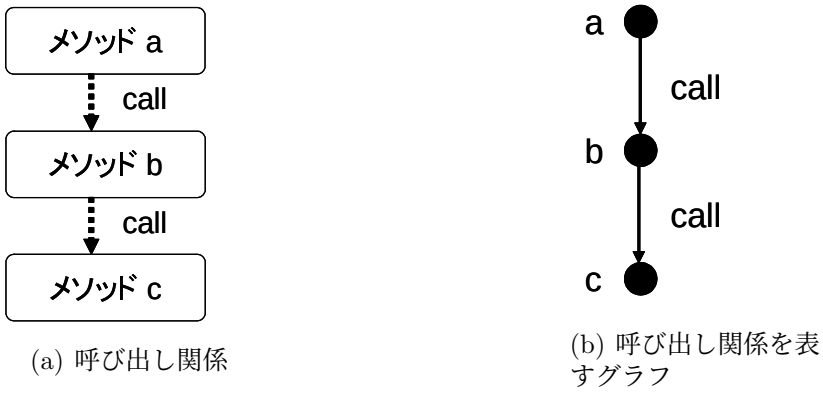
(b) リファクタリング後（1 つのみ）

(c) リファクタリング後（3 つ同時）

図 2.2: クローンセット間に存在する依存関係

図 2.2(a) は，クローンセット間に存在する依存関係を表している．図 2.2(a) 中のクラス A，B にまたがる 3 つのクローンセットの間には，メソッド呼び出し関係やフィールドの利用関係が存在する．これらクローンセットのうち，メソッド a_1 とメソッド b_1 が属するクローンセットのみに対して集約を試みると，親クラスに新たに作成したメソッド s_1 から子クラスのメソッド a_2 と b_2 を呼び出すことが出来なくなる（図 2.2(b)）．ここで，メソッド a_2 とメソッド b_2 が属するクローンセット，加えて，それらメソッドがそれぞれ呼び出しているメソッド a_3 とメソッド b_3 が属するクローンセットについても同様に集約を行えば，メソッド a_1 とメソッド b_1 が属するクローンセットについても容易に集約することができる（図 2.2(c)）．

図 2.2(a) のように，あるクローンセットを集約するためには，他のクローンセッ



(a) 呼び出し関係

(b) 呼び出し関係を表すグラフ

図 2.3: 呼び出し関係と呼び出し関係を表すグラフ

トについても集約を行う必要が生じることがある．しかし，既存のコードクローン検出ツールは，クローンセット間に存在するメソッド呼び出し関係やフィールドの利用関係を検出しないため，開発者はクローンセット間の依存関係を知ることが困難である．

本稿では，図 2.2(a) のようなクローンセットの組み合わせを “チェーンドクローンセット” と呼び，“チェーンドクローンセット” に対するリファクタリング支援手法を提案する．

2.3.2 チェーンドクローン

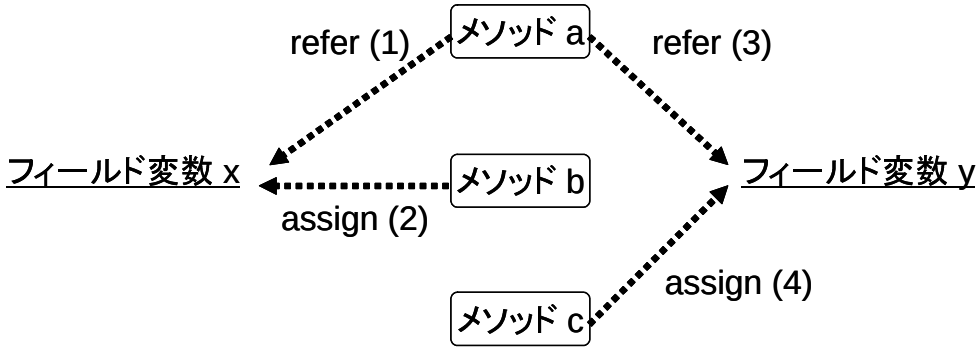
チェーンドクローンを定義するための準備として，メソッドチェーンを定義する．メソッドの集合が与えられたとき，それらメソッド間の依存関係を表す有向グラフが連結グラフになるなら，そのメソッドの集合をメソッドチェーンと定義する．

ここで扱う依存関係は，以下の 2 種類である．

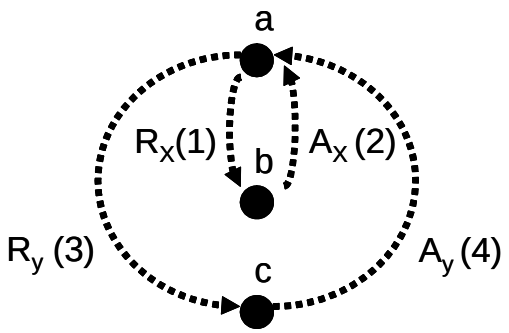
(1) メソッドの呼び出し関係

(2) 同一フィールド変数の共有（参照または代入）

図 2.3(a) は呼び出し関係を含むメソッドチェーンの例である．この例では，メソッド a が b を，メソッド b が c を呼び出している．また，図 2.3(b) は，図 2.3(a) のメソッドチェーンの依存関係をラベル付き有向グラフで表したものである．有向辺に付属しているラベル “call” は，依存関係の種類がメソッドの呼び出し関係であることを表している．例えば，メソッド a が b を呼び出しているとき，有向辺 (a, b) を引き，ラベル “call” を付ける．なお，1 つのメソッドが同一のメソッドを 2 回呼び出している場合は，それらメソッド間に呼び出し関係を表す有向辺を 2 本追加する．



(a) フィールド変数の使用



(b) フィールド変数の共有関係を表すグラフ

図 2.4: フィールド変数の使用と共有関係を表すグラフ（括弧内の数字は対応する関係を表す）

図2.4(a) はフィールド変数を使用しているメソッドチェーンの例である．図2.4(a) の有向辺に付属しているラベル “refer” はフィールド変数の参照を表しており，“assign” はフィールド変数への代入を表している．図2.4(a) のメソッド a と b は，フィールド変数 x を共有しており，メソッド a，c はフィールド変数 y を共有している．また，図2.4(b) は，図2.4(a) のメソッドチェーンに含まれるメソッド間の共有関係をラベル付き有向グラフで表したものである．有向辺に付属しているラベル “$R_x(refer)$” はフィールド変数 x への参照による共有関係，ラベル “$A_y(assign)$” は，フィールド変数 y への代入による共有関係を表している．例えば，メソッド a が変数 x を参照し，かつメソッド b が変数 x を使用（参照または代入）しているとき，有向辺 (a, b) を引き，ラベル “R_x” を付ける．また，メソッド c が変数 y に代入し，かつメソッド a が変数 y を使用しているとき，有向辺 (c, a) を引き，ラベル “A_y” を付ける．

次に，メソッドチェーンを用いてチェーンドクローンを定義する．2つのメソッドチェーンが互いにチェーンドクローンとなるのは，各メソッドチェーンが持つ依存関係のグラフが同形であり，対応する頂点（メソッド）が同一クローンセットに含まれ，対応する辺（依存関係）のラベルは等しいときである．また，互いに

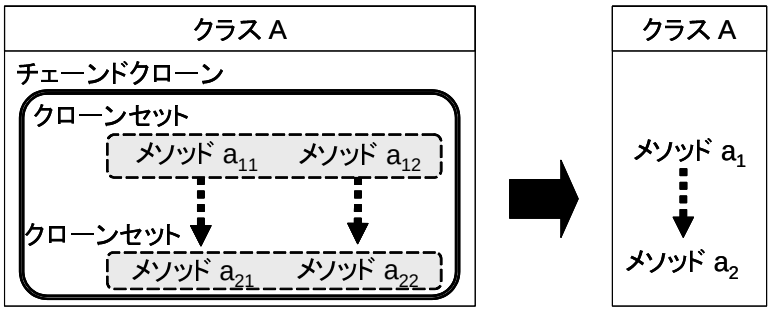


図 2.5: ケース 1

チェーンドクローンであるメソッドチェーンの同値類を，チェーンドクローンセットと呼ぶ.

2.3.3 チェーンドクローンセットに対するリファクタリング

ここでは，チェーンドクローンセットに対して考えられるリファクタリングについて説明するため，適用可能なリファクタリングパターンが異なる 4 つのケースを紹介する.

ケース 1 は，チェーンドクローンセットが 1 つのクラスに包含されている場合である．ケース 1 では，その 1 つのクラス内にクローンセットを集約可能である．図 2.5 は，ケース 1 のチェーンドクローンセットに対するリファクタリングの例である．互いにクローンであるメソッド a_{11} とメソッド a_{12} を集約しメソッド a_1 とし，同様に互いにクローンであるメソッド a_{21} とメソッド a_{22} を集約しメソッド a_2 としている.

ケース 2 は，チェーンドクローンセットが以下の 2 つの条件を満たす場合である.

- チェーンドクローンセットに含まれるメソッドは，全て兄弟クラスに属する.
- 各メソッドチェーンは，それぞれ 1 つのクラスに包含されている.

ケース 2 は，“Pull Up Method” パターンを適用することで，リファクタリングできる．つまり，兄弟クラスにまたがって存在するクローンセットを，親クラスに集約することでリファクタリングできる．図 2.6(a) は，ケース 2 に対するリファクタリングの例である．この例では，兄弟クラスであるクラス A，B にまたがって存在する 2 つのクローンセットを，親クラス S に作成したメソッドに集約している．具体的には，クラス A のメソッド a_1 とクラス B のメソッド b_1 を集約し，親クラス S のメソッド ab_1 とし，同様にクラス A のメソッド a_2 とクラス B のメソッド b_2 を集約し，親クラス S のメソッド ab_2 としている.

ケース 3 は，チェーンドクローンセットが以下の 2 つの条件を満たす場合である.

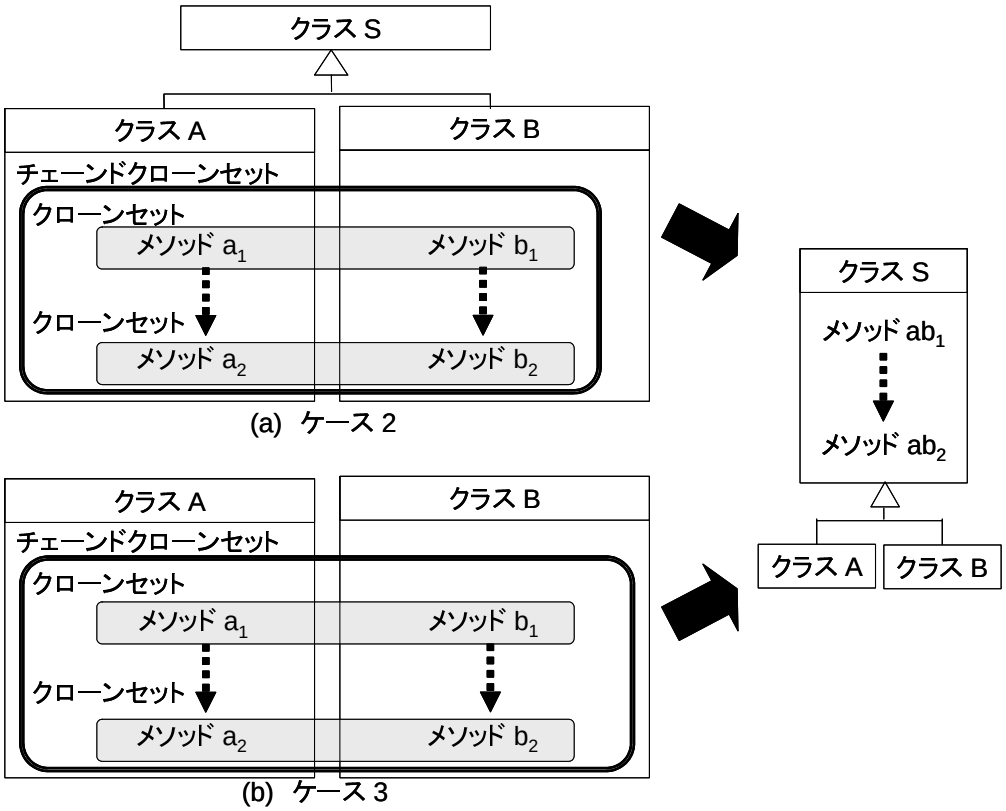


図 2.6: ケース 2 とケース 3

- チェーンドクローンセットに含まれるメソッドを持つクラスは，いずれも共通の親クラスを持たない．
- 各メソッドチェーンは，それぞれ 1 つのクラスに包含されている．

ケース 3 は，“Extract SuperClass” パターンを適用することで，リファクタリングできる．つまり，チェーンドクローンセットに含まれるメソッドを持つクラスに対して，共通の親クラスを作成し，クラス間をまたがって存在するクローンセットを，新たに作成した親クラスに集約することでリファクタリングできる．図 2.6(b) は，ケース 3 に対するリファクタリングの例である．この例では，まず共通の親クラスを持たない 2 つのクラス A, B に，共通の親クラス S を作成している．その後，ケース 2 と同様に兄弟クラスとなったクラス A，B にまたがって存在する 2 つのクローンセットを，親クラス S に作成したメソッドに集約している．

ケース 4 は，チェーンドクローンセットが以下の条件を満たす場合である．

- 各メソッドチェーンは，複数のクラスにまたがって存在する．つまり依存関係が複数のクラス間にまたがっている．

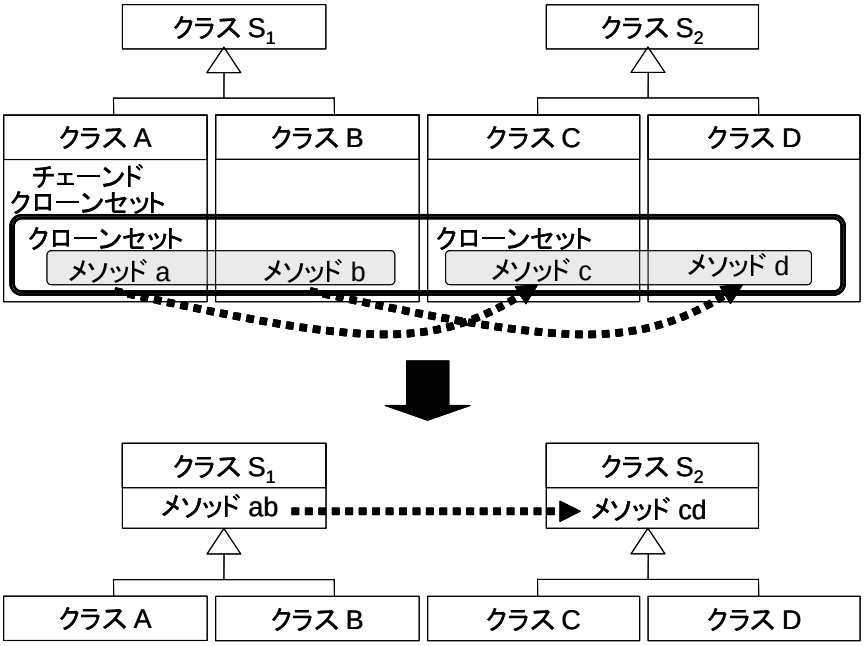


図 2.7: ケース 4

ケース 4 は，チェーンドクローンセット単位でリファクタリングできない場合である．だが，チェーンドクローンセットを複数のクローンセットとして扱い，それぞれの親クラスに集約することでリファクタリングできるため，クローンセット単位でのリファクタリングを検討すべきである．図 2.7 は，ケース 4 にクローンセット単位でのリファクタリングを適用した例である．この例では，兄弟クラスであるクラス A，B にまたがって存在する 2 つのクローンセットを，それぞれの親クラスに作成したメソッドに集約している．具体的には，クラス A のメソッド a とクラス B のメソッド b を集約し，親クラス S_1 のメソッド ab とし，同様にクラス C のメソッド c とクラス D のメソッド d を集約し，親クラス S_2 のメソッド cd としている．

2.3.4 チェーンドクローンセットの分類

前節の 4 つのケースのように，チェーンドクローンセットを分類する．前節の 4 つのケースには，それぞれ適合するための条件があった．それらは，次の 2 つである．

C1 チェーンドクローンセットに含まれるメソッドが所属するクラス間の関係についての条件

C2 メソッドチェーンに含まれるメソッドが所属するクラス間の関係についての条件

ここでのクラス間の関係とは，クラス階層上の関係のことである．クラス間の関係は，次に 3 つに分類できる．

R1 全ての同一クラス

R2 共通の祖先クラスを持つ

R3 共通の祖先クラスを持たないクラス

条件の種類とクラス間の関係を組み合わせることにより，チェーンドクローンセットを表 2.1 のように分類できる．

次の 4 つの分類については，以下に示すリファクタリングを行うことできると考えられる．

分類 1 前節のケース 1 である．図 2.5 の例のように，チェーンドクローンセットを包含しているクラスに，全てのクローンセットを集約することができる．

分類 2 前節のケース 2 である．図 2.6(a) の例のように，“Pull Up Method” パターンを適用できる．

分類 3 前節のケース 3 である．図 2.6(b) の例のように，“Extract SuperClass” パターンを適用できる．

分類 4 前節のケース 4 である．図 2.7 からわかるように，チェーンドクローンセット単位でのリファクタリングを行うことが出来ないが，クローンセット単位でのリファクタリングを検討すべきである．

2.3.5 チェーンドクローンセットの分類を目的としたメトリクス

ここでは，チェーンドクローンセットの分類を行うためのメトリクスを 2 つ提案する．1 つは C1 を評価するメトリクス，もう 1 つは C2 を評価するメトリクスである．これらメトリクスは，メソッド間のクラス階層上における関係を表す．こ

表 2.1: チェーンドクローンセットの分類

<table>
<tr><th>C1 \ C2</th><th>R1</th><th>R2</th><th>R3</th></tr>
<tr><td>R1</td><td>分類 1</td><td colspan="2" rowspan="3">分類 4</td></tr>
<tr><td>R2</td><td>分類 2</td></tr>
<tr><td>R3</td><td>分類 3</td></tr>
</table>

の関係は，2.2 節で述べた $DCH(S)$ メトリクス（クローンセット S に含まれる各コード片間のクラス階層内における最大の距離）によって表すことができる．

まず，$DCH(S)$ メトリクスを用いて，C1 を表す **DCHS(T)** メトリクスを定義する．チェーンドクローンセット T を n 個のクローンセット $S_1, S_2, \cdots, S_n$ に分割する (すなわち，$S_1 \cup S_2 \cup \cdots \cup S_n = T, S_i \cap S_j = \emptyset$, $1 \leq i \leq n$, $1 \leq j \leq n$, $i \neq j$)．更に，クローンセット S_i には，複数のメソッドが含まれるとする．このとき，$DCHS(T)$ メトリクスの定義は以下のようになる．

$$DCHS(T) = max\{DCH(S_1), \cdots, DCH(S_n)\}$$

同様に $DCH(S)$ メトリクスを用いて，C2 を表す **DCHD(T)** メトリクスを定義する．チェーンドクローンセット T 中には，n 個のメソッドチェーン $M_1, M_2, \cdots, M_n$ が含まれるとする．更に，メソッドチェーン M_i には，複数のメソッドが含まれるとする．このとき，$DCHD(T)$ メトリクスの定義は以下のようになる．

$$DCHD(T) = max\{DCH(M_1), \cdots, DCH(M_n)\}$$

図 2.8 は，提案する 2 つのメトリクスの算出例である．ここでは，分類 2 のチェーンドクローンセットを例として用いる．このチェーンドクローンセットには，クラス A，B，C にまたがって 2 つのクローンセット S_1，S_2 が存在する．各クローンセットについてそれぞれ $DCH(S_1)$，$DCH(S_2)$ を求めると，クラス A，B，C は共通の直接の親クラス S を持っているため両者とも 1 になる．$DCHS(T)$ の値は，これらの最大値の 1 である．また，このチェーンドクローンセットには，3 つのメソッドチェーン M_1, M_2, M_3 が含まれている．各メソッドチェーンについてはそれぞれ $DCH(M_1)$，$DCH(M_2)$，$DCH(M_3)$ を求めると，各メソッドチェーンはそれぞれ 1 つのクラスに包含されているため全て 0 になる．よって，$DCHD(T)$ の値は，これらの最大値の 0 である．

2.3.6 実装

提案手法を Aries のコンポーネントの 1 つとして実装した．具体的には，Aries に対して，以下の 3 つの機能を追加した．

(F1) チェーンドクローンセットの検出機能

(F2) 提案したメトリクスの算出機能

(F3) チェーンドクローンセットおよびメトリクス値の表示機能

(F1) は，まず CCFinder および Aries を用いて各クローンセットが含むコード片を検出する．次に，それらコード片を対象に，メソッド呼び出し関係と変数の共

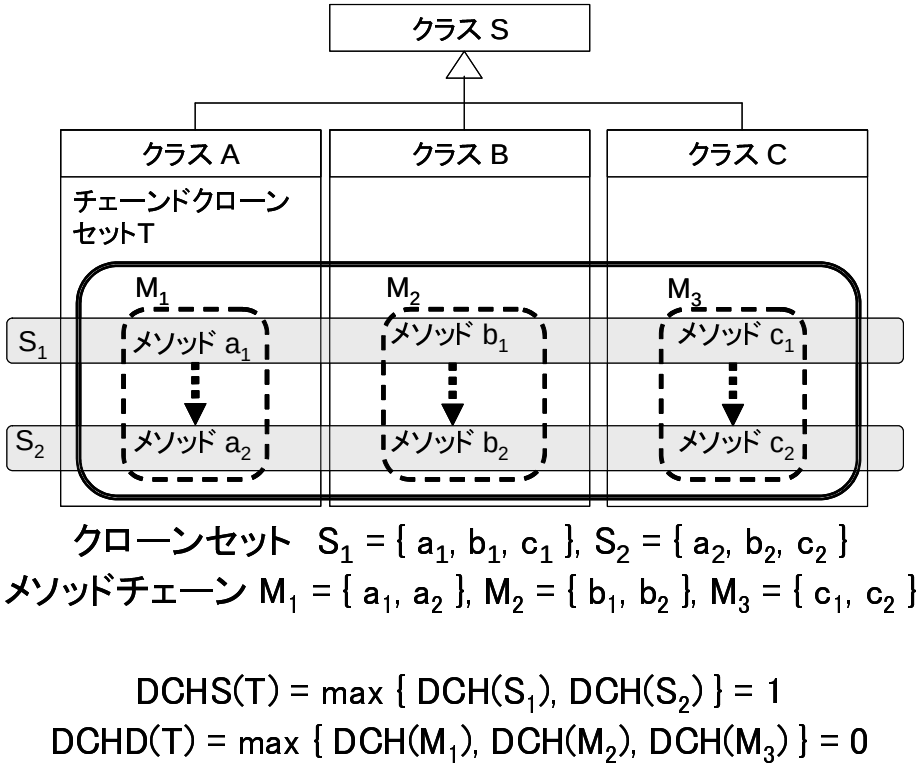


図 2.8: 提案するメトリクスの算出例

有関係を表すグラフを構築する．その後，構築したプログラム依存グラフに含まれる部分グラフから同形グラフを検出することにより，チェーンドクローンセットを検出する．(F2) は，Aries の $DCH(S)$ メトリクスを計算する機能を拡張した．(F3) を実現するために，以下の 3 つのビューを 3 つ追加した．

Chained Clone Set Selection View

図 2.9 は，*Chained Clone Set List* のスナップショットである．

この画面には，検出されたチェーンドクローンセットの一覧が表示される．各チェーンドクローンセットは，分類毎に表示される．また，各チェーンドクローンセットに対して，2.3.5 節で提案した $DCHS(T), DCHD(T)$ メトリクスや，CCFinder が検出する $LEN(S), POP(S), DFL(S)$ メトリクス [39] の値がそれぞれ表示されている．

ユーザはこの画面で，各分類に所属しているチェーンドクローンセットの数や規模を知ることができる．また，関心のあるチェーンドクローンセットを選択すると，*Chained Clone Set View* が開き，そのチェーンドクローンセットの詳細を確認することができる．

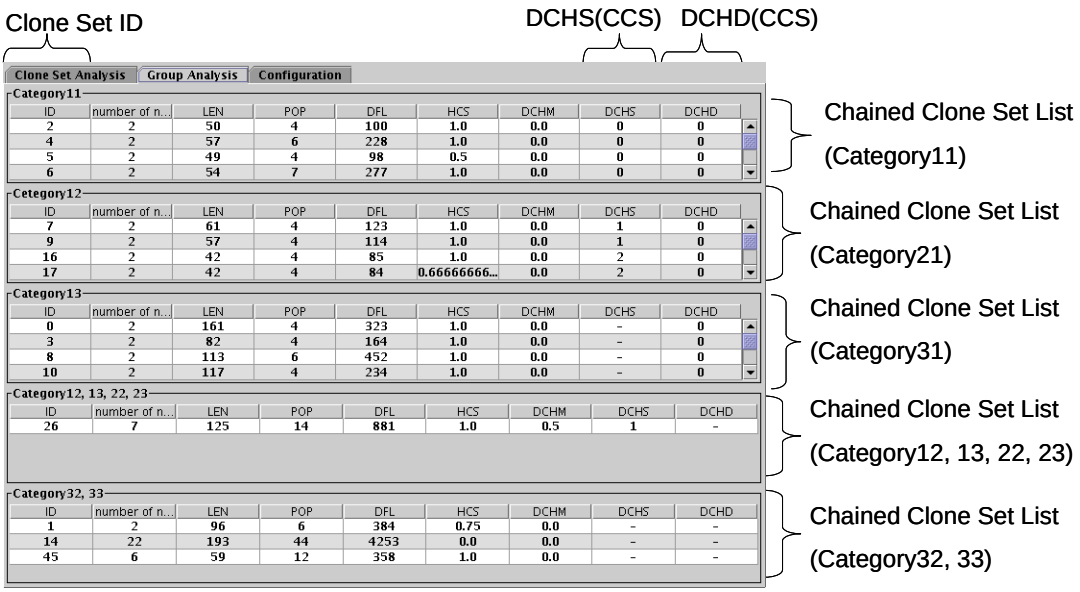


図 2.9: Chained Clone Set Selection View

Chained Clone Set View

図 2.10 は，*Chained Clone Set View* のスナップショットである．

この画面には，*Chained Clone Set List* で選択したチェーンドクローンセットの詳細が表示される．この画面は，チェーンドクローンセットに含まれているクローンセットのリストである *Clone Set List*，各コード片とチェーンドクローンセット内に存在する依存関係を表示する *Depedence Relation View* の二つから構成される．

ユーザはこの画面で，選択したチェーンドクローンセットにどのクローンセットが含まれているか，またどのような依存関係が含まれているかを確認することができる．また，関心のあるクローンセットを選択すると，*Source Code View* が開き，含まれるコード片を閲覧することができる．

Source Code View

図 2.11 は，*Source Code View* のスナップショットである．*Chained Clone Set View* において選択されたコードクローンのソースコードが表示されている．*Variable List* は，選択されたメソッド中で使用している変数のリストである．

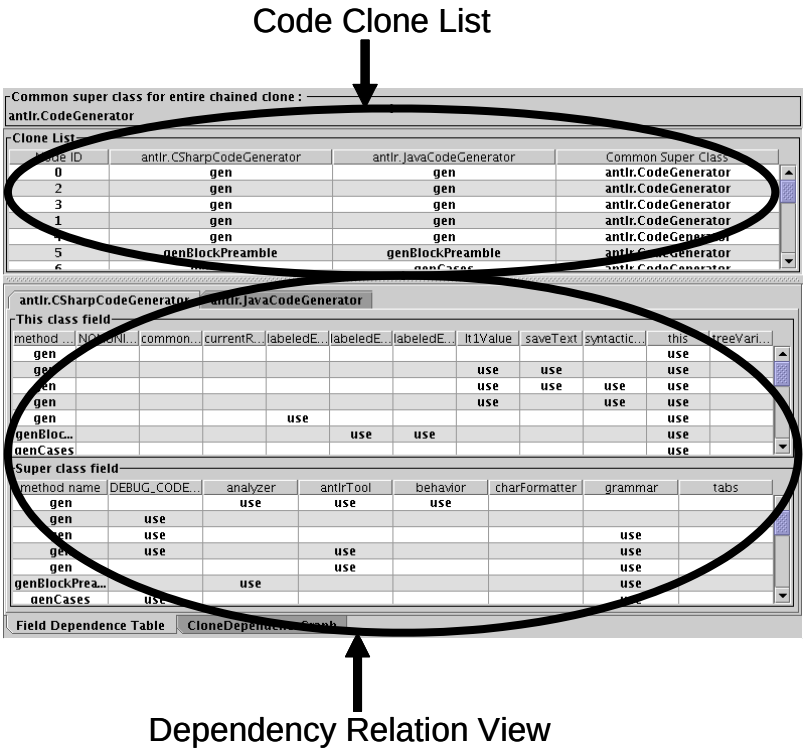


図 2.10: Chained Clone Set View

2.4 適用実験

2.4.1 概要

提案手法の有効性を確かめるため，ケーススタディを行った．具体的には，以下の 2 つを確認した．

- クローンセット単位の検出と比較して，検出できたチェーンドクローンセットの規模が大きいか
- クローンセット単位でのリファクタリングと比較して，容易にリファクタリングできているか

なお，この章におけるチェーンドクローンセットは，極大チェーンドクローンセット[1]を指す．

適用対象は，次の 2 つのオープンソースソフトウェアである．

- ANTLR 2.7.4[3](4.7 万行，285 クラス)

[1]与えられたチェーンドクローンセットを真に包含する如何なるチェーンドクローンセットも存在しないとき，そのチェーンドクローンセットを極大チェーンドクローンセットと呼ぶ．

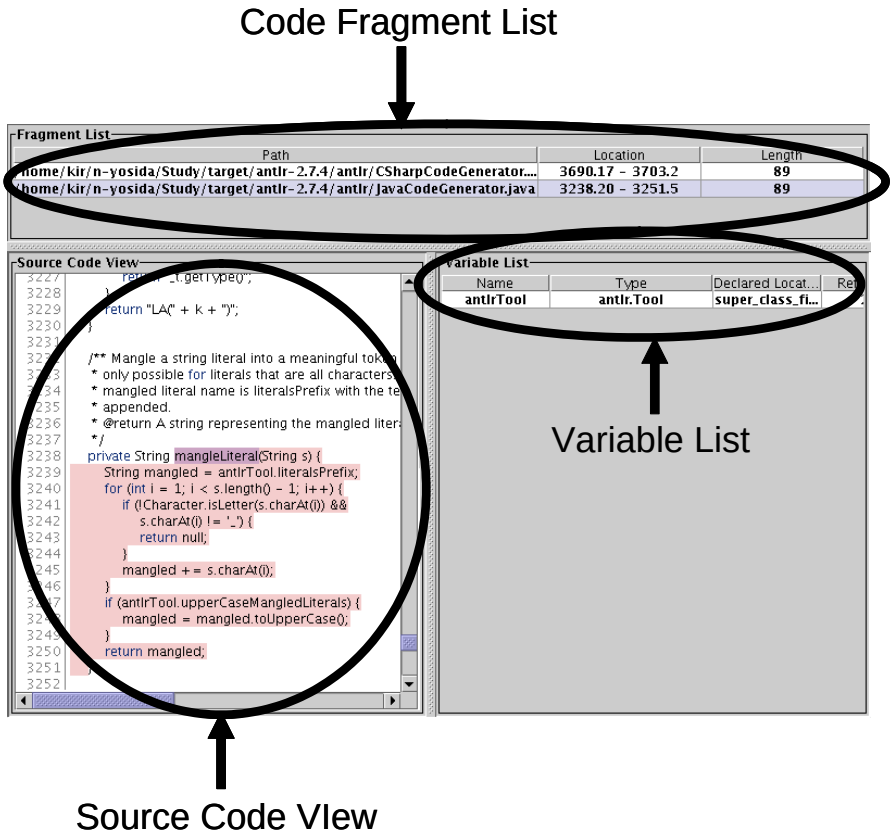


図 2.11: Source Code View

- JBoss 3.2.6[36](64 万行， 3364 クラス）

ANTLR は，3 つのプログラミング言語（Java，C#，C++）に対応したコンパイラ・コンパイラである．JBoss は，J2EE アプリケーションサーバである．

2.4.2 チェーンドクローンセットの検出

前述の 2 つのソフトウェアに対し，提案手法に基づくチェーンドクローンセットの検出，および従来手法に基づくクローンセットの検出を行った．提案手法ではメソッド単位のコードクローンのみを扱うため，従来手法に基づく検出でもメソッド単位のクローンセットのみを対象とした．また，CCFinder が検出するコードクローンの最小トークン数は 30 に設定した．

検出結果の比較を行うために，2 つの評価基準として，メソッド数と，メソッド行数を用いる．ここで，メソッド数は，検出単位毎（クローンセット毎やチェーンドクローンセット毎）に含まれるメソッド数を求め，総クローンセット数や各分類に属する全てのチェーンドクローンセット数で除算した値とした．メソッド

行数は，検出単位毎に最長メソッドの行数を求め，総クローンセット数や各分類に属する全てのチェーンドクローンセット数で除算した値とした．

設定した評価基準に基づいて，クローンセットの検出結果（表2.2）と分類1，2，3 に属したチェーンドクローンセットの検出結果（表2.3(a)）を比較する．

まず，ANTLR では分類2 に属したチェーンドクローンセットのメソッド数やメソッド行数が極めて大きかった．分類2 のメソッド数はクローンセットの 8.3 (19/2.3) 倍，メソッド行数は5.0 (54.0/10.8) 倍であった．一方，分類1，3 に属したチェーンドクローンセットのメソッド数は両者ともクローンセットの1.7 (4.0/2.3) 倍，メソッド行数はそれぞれ2.6 (27.7/10.8) 倍，3.2 (35.0/10.8) 倍であった．分類2 に属するチェーンドクローンセットの多くは，図2.12 のような，Java，C#，C++に対応した出力を行う箇所から検出された．これら言語に対応した出力処理は類似しており，大量のコードクローンを含んでいた．

次に，JBoss の結果を見てみると，分類1，2，3 に属したチェーンドクローンセットのメソッド数はそれぞれクローンセットの 1.8 倍〜2.8 倍，メソッド行数がそれぞれ2.4 倍〜3.2 倍であった．これらの結果から，チェーンドクローンセットの規模がクローンセットに比べて大きいことがわかる．続いて，分類1，2，3 に属したチェーンドクローンセットの検出結果（表2.3(a)）と，分類4 に属したチェーンドクローンセットの検出結果（表2.3(b)）を比較する．2.3.4 節で述べたように，表2.3(a) で示した分類1，2，3 は，チェーンドクローンセット単位でのリファクタリング可能であるが，表2.3(b) で示した分類4 はチェーンドクローンセット単位でのリファクタリングが出来ない．分類1，2，3 と比較して分類4 に属するチェーンドクローンセットは少ないことがわかる．特に，ANTLR からは検出されなかった．一方，JBoss では4 種類のチェーンドクローンセットが検出された．これらのメソッド数, メソッド行数は，分類1〜3 に比べて数倍の大きさになっている．

2.4.3 チェーンドクローンセットに対するリファクタリングの例

ANTLR から検出された全てのチェーンドクローンセットと JBoss から検出された8 つのチェーンドクローンセットを対象として，提案手法に基づくリファクタリングを行った．ここでは，それらの中から2 つの例を紹介する．

まず，図2.12 ，図2.14 で示すチェーンドクローンセットに対し，提案手法により提示されたリファクタリングパターンを適用できることを確認した．図2.12 は分類2 であるから，“Pull Up Method” パターンを適用した．その結果，図2.13 のようになった．ANTLR パッケージ中の examples ディレクトリ以下にある全てのテストケース（計86 ファイル，文法ファイル）を用いて回帰テストを行い，外部的振る舞い（出力結果）が変化していないことを確認した．また，図2.14 は分類3 であるから，“Extract SuperClass” パターンを適用した．その結果，図2.15 のようになった．JBoss パッケージの testsuite ディレクトリ以下にある全テストケー

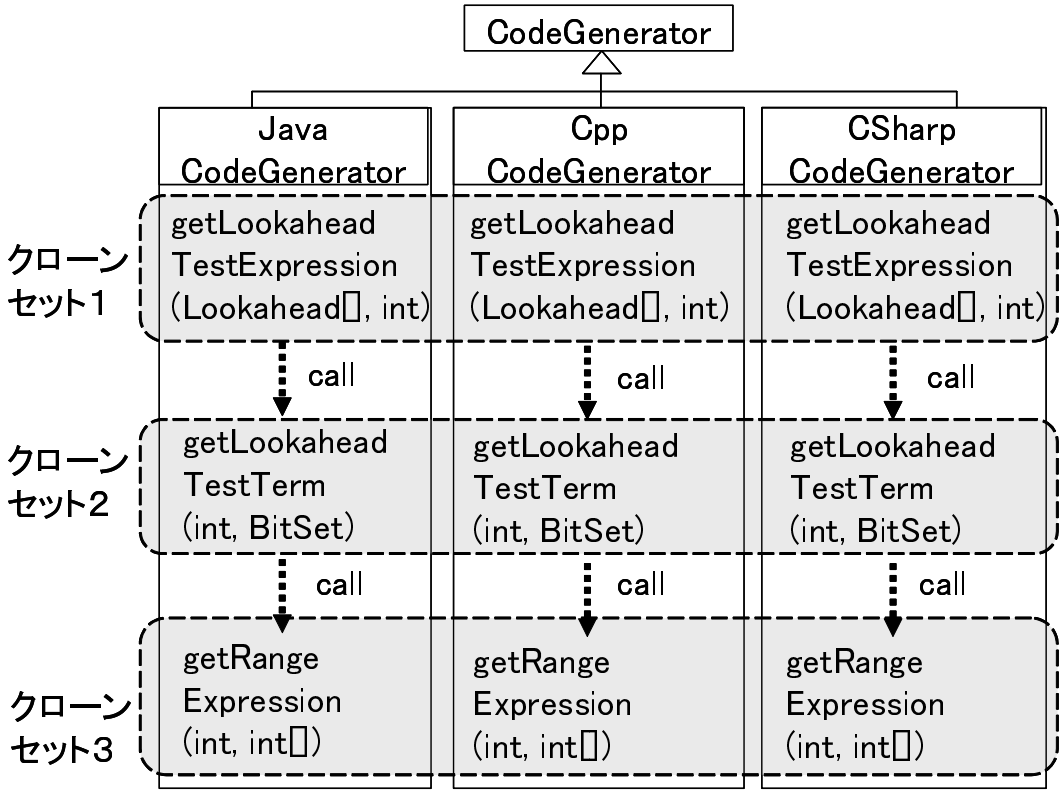


図 2.12: ANTLR から検出されたチェーンドクローンセットの例

ス（計 65 ファイル，JUnit フレームワーク [38] を使用）を用いて回帰テストを行い，外部的振る舞いが変化していないことを確認した．

更に，従来手法に基づいて，チェーンドクローンセットを構成するクローンセットに対し集約を試みると，工夫が必要となる場合があることを確認した．具体的には，図 2.12，図 2.14 からそれぞれクローンセットを 1 つ選び，従来手法により提示されたリファクタリングパターンの適用をした．まず，図 2.12 のクローンセット 1 に対し集約を試みると，提示されたリファクタリングパターンの適用に加えて，クローンセット 2 のメソッドに対応する抽象メソッドを CodeGenerator クラスに追加する必要があった．なお，クローンセット 2 に対して集約を試みた場合も同様であることが確認できた．また，図 2.14 のクローンセット 2 を新たに作成した親クラスに集約を試みると，提示されたリファクタリングパターンの適用に加えて，クローンセット 2 のメソッドに対応する抽象メソッドを親クラスに追加する必要があった．

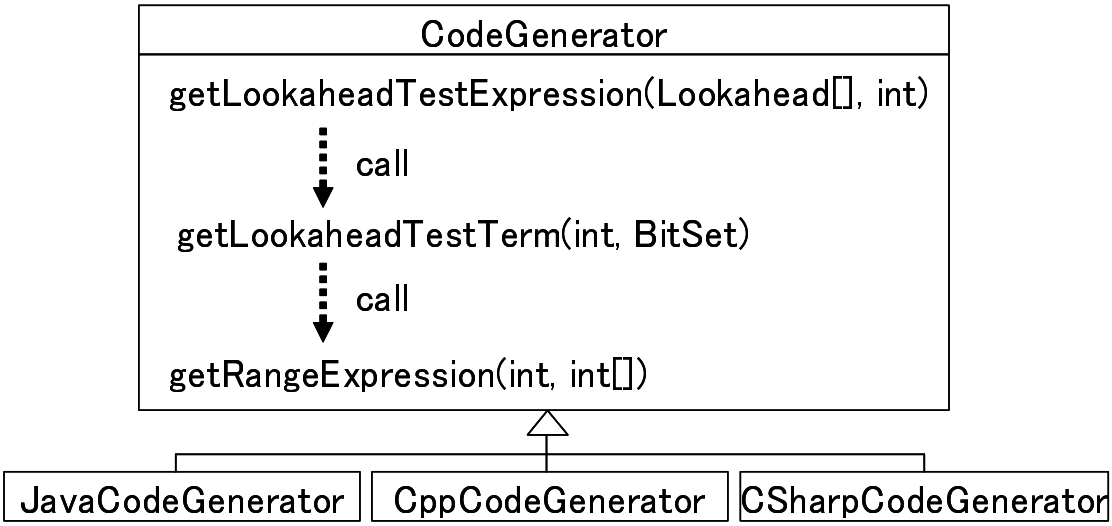


図 2.13: 図 2.12 のチェーンドクローンセットをリファクタリングした例

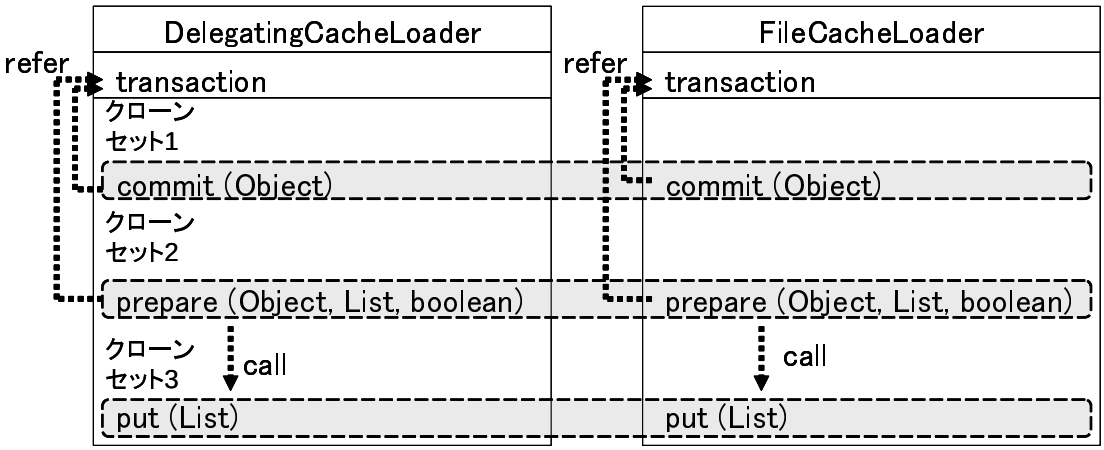


図 2.14: JBoss から検出されたチェーンドクローンセットの例

2.4.4 考察

ここでは，今回のケーススタディに基づいて，提案手法の妥当性・制限等について考察を行う．

(1) **対象ソフトウェア** ケーススタディでは，Java 言語で開発された 2 つのオープンソースソフトウェアを対象として有効性の確認を行った．オブジェクト指向型言語であれば，Java 言語以外で開発されたソフトウェアであっても適用可能であると考えられる．なぜなら，Java 言語以外のオブジェクト指向型言語で開発されたソフトウェアであっても，リファクタリングや依存関係解析を適用可能だからである．今後，Java 言語以外で開発されたソフトウェアや

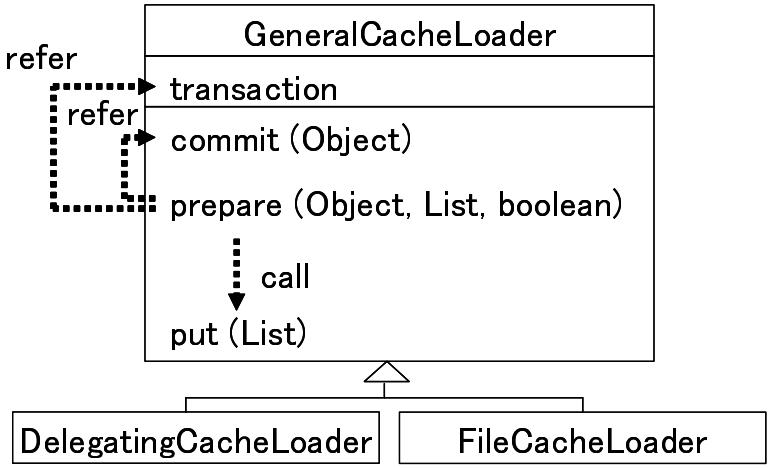


図 2.15: 図 2.14 のチェーンドクローンセットをリファクタリングした例

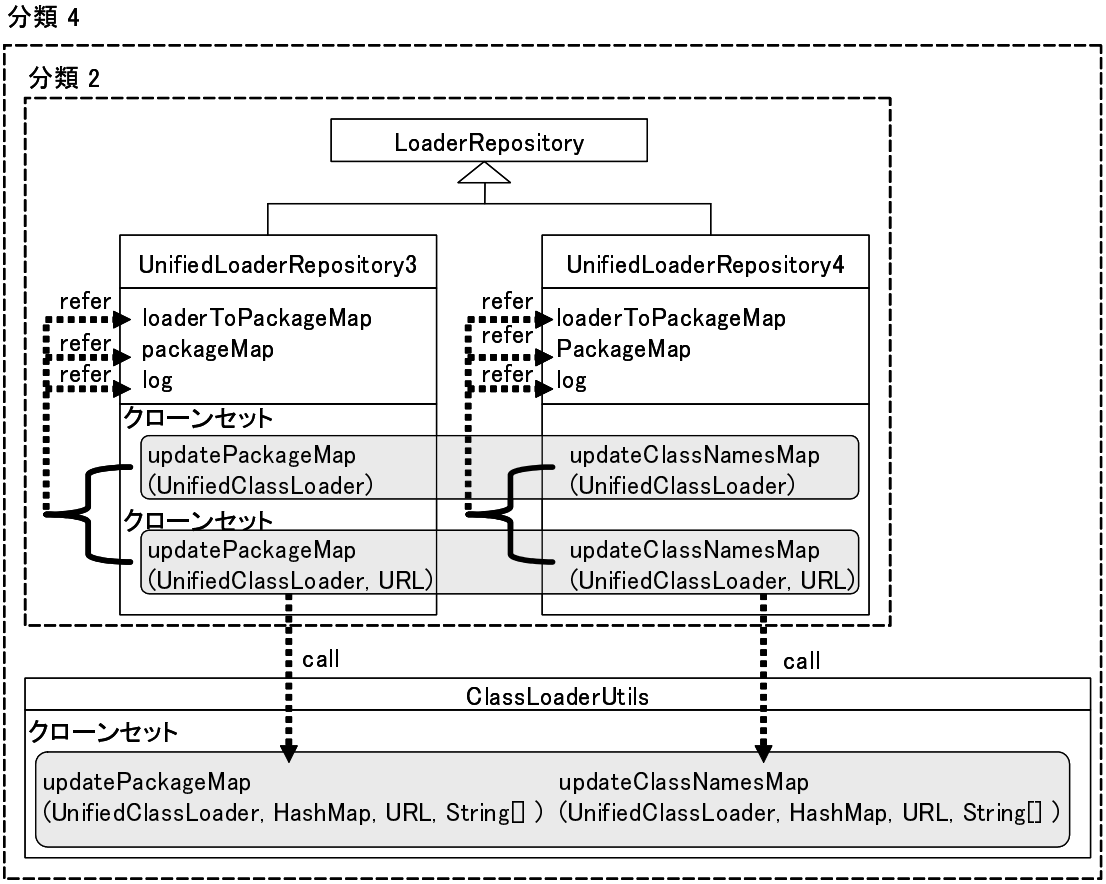


図 2.16: 分類 4 のチェーンドクローンセットの例

商用ソフトウェアなど，様々な種類のソフトウェアを対象に有効性の評価を行う必要がある．

(2) **被支援者の知識，経験** 今回のケーススタディにおけるリファクタリング作業は，全て著者が行った．そのため，手法の詳細を知らない人やリファクタリング経験が少ない人に対しても，有効な支援が行えるかどうか評価する必要がある．例えば，一般の開発者に本稿のツールを使用してもらい，リファクタリングにかかった時間を計測することで効率を評価するということが考えられる．

(3) **対象とするコード片や依存関係の種類** 提案手法は，メソッドより小さい単位のコード片からなるクローンセットは対象としていない．また，メソッドの呼び出し関係および変数の共用による依存関係のみを扱っているため，その他のデータ依存関係や制御依存関係を対象としていない．今後，対象とするコード片や依存関係を増やすことで検出可能なチェーンドクローンセットの規模を大きくし，有効性の評価を行う必要がある．

(4) **分類 4 のチェーンドクローンセットへの支援** JBoss から分類 2 のチェーンドクローンセットを内包する分類 4 のチェーンドクローンセットが検出された（図 2.16）．提案手法は，これに対しクローンセット単位でのリファクタリングを提示するが，内包された分類 2 のチェーンドクローンセットに “Pull Up Method” リファクタリングパターンを適用可能である．このような場合は，内包されたチェーンドクローンセットに対してリファクタリングパターンの提示を行うべきであると考えられる．

2.5 関連研究

CCFinder や Balazinska[10] らの手法を用いることにより，クローンセットの検出を行うことはできる．本稿の手法では，チェーンドクローンセット単位でのリファクタリングを提示することにより，大規模なリファクタリングを実現することが出来た．また，CCFinder が検出するクローンセットには容易にリファクタリングできないものが含まれており，Balazinska らの手法も同様と考えられる．本稿では，それら容易にリファクタリングできないクローンセットを組み合わせることで容易にリファクタリングできる場合があることを示し，それらクローンセットに対するリファクタリング手法を提案した．

Komondoor[42] らの手法は，プログラムスライシング技術を用いて，ソースコードからプログラム依存グラフを構築し，そのグラフ上で同形である箇所をコードクローンとして検出している．よって，本稿で用いている CCFinder が検出できないコードクローン（一部の文の出現順序が異なっている reodered clone 等）を検出することができる．しかし，プログラム依存グラフの構築にかかる計算コストは

非常に大きいため，大規模ソフトウェアへの適用は現実的でない．本稿の手法は，CCFinder が検出したコードクローンに対して，メソッド呼び出し関係と変数の利用関係のみを解析しているため，ケーススタディで示した規模のソフトウェアに適用可能である．実際に，ANTLR を対象としたチェーンドクローンセットの検出を約 1 分 4 秒で行うことができた[2]．

2.6 結論

本稿では，クローンセットに含まれるメソッド間の依存関係に着目し，チェーンドクローンセットを定義した．そして，チェーンドクローンセットに対しリファクタリングパターンを提示するためのメトリクスを提案した．最後に，提案手法をリファクタリング支援ツールとして実装し，2 つのオープンソースソフトウェアに適用することで，有効性の評価を行った．有効性の評価として，チェーンドクローンセットの規模がクローンセットと比べて大きいこと，およびチェーンドクローンセットのリファクタリングがクローンセット単位のリファクタリングと比べて容易であることを確認した．

今後の課題としては，様々なソフトウェアを対象とした有効性の評価，チェーンドクローンセットの中に異なる分類のチェーンドクローンセットが包含されている場合への対処，対象とする依存関係やコード片の拡大が挙げられる．

[2]実行環境: CPU Xeon 2.80GHz，メモリ 2.5GB，OS FreeBSD 6.1-RELEASE

表 2.2: クローンセットの検出結果

検出数		メソッド数		メソッド行数	
ANTLR	JBoss	ANTLR	JBoss	ANTLR	JBoss
152	377	2.3	2.4	10.8	6.63

表 2.3: チェーンドクローンセットの検出結果

(a) 分類 1，2，3

分類	検出数		メソッド数		メソッド行数	
	ANTLR	JBoss	ANTLR	JBoss	ANTLR	JBoss
1	3	16	4.0	5.8	27.7	16.2
2	6	17	19	4.5	54.0	17.1
3	1	13	4.0	6.8	35.0	21.5

(b) 分類 4

検出数		メソッド数		メソッド行数	
ANTLR	JBoss	ANTLR	JBoss	ANTLR	JBoss
0	4	-	19	-	54.5

第3章　類義語の特定に基づく類似コード片検索法

3.1　導入

ソフトウェア保守を困難にする要因の1つとして**類似コード片**が指摘されている [7, 13, 37, 39, 42, 46, 47, 66]．類似コード片とは，ソースコード中の**コード片**（ソースコードの一部分）のうち，一致もしくは類似した要素（識別子や構文など）を含むコード片を持つものを指す．類似コード片は，既に開発されたコード片のコピーとペーストによる再利用や定型処理の実装などが理由で作成される [13, 41]．特に，Linux や JDK (Java Development Kit) などの大規模ソースコードは大量の類似コード片を含むことが報告されている [39][47]．

ソフトウェアの保守を行う際に，あるコード片を修正するとその全ての類似コード片を見つけ出し修正を行う必要が生じることがある．特に，ソースコード中に欠陥が見つかった場合には，その欠陥を含むコード片の類似コード片を探し，検査する必要がある [46][47][66]．しかし，ソフトウェア中の類似コード片を人手で探すためには大きな労力が必要となる．特に，大規模ソフトウェアが対象の場合，全ての類似コード片を人手で探すことはより困難となる．

類似コード片の検索に用いることができる方法として，キーワード検索やコードクローン検出法が挙げられるが，両者とも効果的な検索を行うことができるクエリ（検索質問）を与えることは難しい．キーワード検索を用いる場合，修正を要するコード片から抽出したキーワードを grep[26] などのツールに与えることで，キーワードを含むコード片を列挙する．しかし，対象ソフトウェアを十分に理解した開発者でなければ，適切なキーワードを抽出することが困難である（**問題点 1**）．その要因の1つとして，grep 等のツールは文字列が完全に一致するコード片のみを出力するため，わずかに異なる文字列を含むコード片であっても検索結果に含めることはできないことが挙げられる．一方，コードクローン検出法を用いる場合，コードクローン検出ツールを用いて修正を要するコード片とトークン列が等価なコード片を列挙する [30]．しかし，トークン列に些細な差異（例外処理の有無など）があるコード片を列挙することはできない（**問題点 2**）．

本研究では，容易にクエリを与えることができ，かつクエリと些細な差異がある部分であっても提示できる手法を提案する．提案する手法は，コード片をクエリとして与えると，識別子の類似性に基づいて対象ソースコードから**類似関数**（ク

エリとして与えられたコード片の類似コード片を 1 つ以上含む関数）を検索する．具体的には，まず自然言語処理の分野で提案されている Dagan らの手法 [19] を用いて，**語**（識別子を分割・正規化した後の文字列）の類義語を特定する（3.3.2 節参照）．次に，**入力コード片**（クエリとして与えられたコード片）に含まれる全ての語と一致する，もしくは類義語である語を含む関数を検出し，類似関数として提示する．

提案手法は，以下の 3 つの特徴を持っている．

- 修正を要するコード片を入力コード片として与えるのみで検索を行うことができる．**問題点 1** で述べたように，grep を用いる場合，適切なキーワードをコード片から抽出する必要がある．
- 入力コード片と類似した処理を表す関数の一部に異なる識別子が含まれていたとしても，類義語を特定することで類似関数として提示できる可能性がある．**問題点 1** の要因の 1 つとして挙げたように，grep を用いると完全に一致する文字列を含む部分のみが出力される．
- コード片が含む識別子の類似性を判定するため，トークン列に些細な差異（例外処理の有無など）がある関数であっても類似関数として提示できる可能性がある．**問題点 2** で述べたように，コードクローン検出法はこのような関数を検出できない．

提案手法を用いて，開発者が類似関数を検索し，確認する手順を以下に示す．

1. 対象ソースコード中から入力コード片を抽出する．
2. 類義語の特定に用いる閾値を提案手法に入力する．提案手法は，入力された閾値より距離が小さい語同士を類義語とする（詳しくは，3.3.2 節参照）．
3. 提案手法に入力コード片を与えることで，類似関数を検索する．
4. 検索結果に含まれた関数が多すぎる，もしくは少なすぎる場合は (2) に戻り，閾値を再入力する．関数が多すぎる場合には閾値を小さくし，少なすぎる場合には閾値を大きくする．
5. 検索結果に含まれた関数を一つ一つ確認する．

適用実験では，類似した欠陥を含む関数の検索に対する提案手法の有効性を確認した．具体的には，類似した欠陥（バッファオーバーフローエラーや型キャストの欠如）を複数含むソースコードを対象として，欠陥を含むコード片のうちの 1 つを入力コード片として提案手法に与えたときに，類似した他の欠陥が提示されるかどうか確認した．その結果，提案手法は欠陥を含むコード片を 1 つ入力コードとして与えるだけで，類似した欠陥のうちの多くを提示できることが確認でき

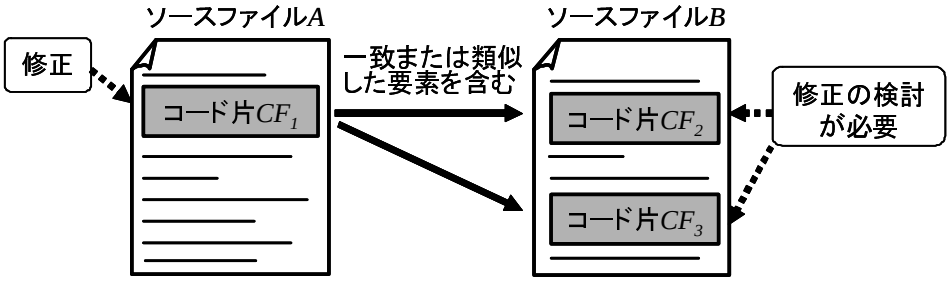


図 3.1: 類似コード片

た．また，grep やコードクローン検出ツール CCFinder[39] を用いて同様の実験を行い提案手法の実験結果と比較することで，それぞれの検索結果が持つ特徴を確認した．

以降，3.2 節では提案手法を説明するための準備として，類似コード片検索について述べ，3.3 節では提案手法である識別子の類似性に基づく類似コード片検索法について述べる．3.4 節では類似した欠陥を含むコード片の検索に提案手法を適用した結果について述べ，3.5 節では 3.4 節で述べた結果を踏まえ考察を行う．3.6 節では関連する手法を提案している研究について議論し，最後に 3.7 節でまとめと今後の課題について述べる．

3.2 類似コード片検索

本稿では，**類似コード片検索**を “ソースコード中から，クエリ（検索質問）として与えられたコード片と一致もしくは類似した要素（識別子や構文など）を含むコード片を提示すること” と定義する．コード片は 5 項組 (ソースファイルを一意に識別できる番号, 開始行, 開始桁, 終了行, 終了桁) で表現する．なお，“コード片 CF と一致もしくは類似した要素を含むコード片” を単に**コード片 CF の類似コード片**と呼ぶ．また，“コード片 CF の類似コード片を 1 つ以上含む関数” を単に**コード片 CF の類似関数**と呼ぶ．

図 3.1 は，修正を要するコード片 CF_1 と，同様の修正を要する類似コード片 CF_2, CF_3 を表している．コード片 CF_1 に含まれる情報を基にクエリを作成し，類似コード片検索を実現したシステムに与えると，その類似コード片 CF_2, CF_3 を提示する．

図 3.2 の 2 つのコード片は，日本語入力システム “かんな”[71] バージョン 3.6 のソースコードに含まれていた類似コード片である．これらコード片は，共にバッファオーバーフローエラーを含んでいる．具体的には，各コード片の 3〜5 行目がバッファからの読み込み処理を表しており，これら処理中にバッファオーバーフローエラーを引き起こす可能性がある[1]．そのため，一方のコード片を基にクエリ

[1] “かんな” バージョン 3.6p1 では，これら欠陥は修正されていた．図 3.3 は，図 3.2(a) の欠陥修

```
ir_debug( Dmsg(10, "ProcWideReq7 start!!¥n") );

buf += HEADER_SIZE; Request.type7.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.number = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.yomilen =(short)S2TOS(buf);
```

バッファからの読み込み処理

(a) 欠陥を含むコード片

```
ir_debug( Dmsg(10, "ProcWideReq14 start!!¥n") );

buf += HEADER_SIZE; Request.type14.mode = L4TOL(buf);
buf += SIZEOFINT;   Request.type14.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type14.yomi = (Ushort *)buf;
```

バッファからの読み込み処理

(b) 同様の欠陥を含むコード片

図 3.2: 類似コード片間の差異

```
ir_debug( Dmsg(10, "ProcWideReq7 start!!¥n") );

if (Request.type7.datalen != SIZEOFSHORT * 3)
        return( -1 );
buf += HEADER_SIZE; Request.type7.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.number = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.yomilen = (short)S2TOS(buf);
```

バッファオーバーフローを検知

バッファからの読み込み処理

図 3.3: 図 3.2(a) のコード片に対する修正

を作成し類似コード片検索を行ったなら，もう一方のコード片が検索結果に含まれることが望ましい．

3.2.1 grep を用いた類似コード片検索

一般の開発者は，grep[26] を用いて類似コード片検索を行うと考えられる．開発者が grep を用いて修正を要するコード片の類似コード片を検索する手順を以下に示す．

1. 修正に関連すると思われるキーワード（行，式，識別子など）を抽出する．
2. そのキーワードを引数として grep を実行する（図 3.4）．
3. grep の出力結果を基に，キーワードを含むコード片を特定する．

grep を用いた方法の問題点は，(1) で適切なキーワードを抽出し検索を行わなければ，効果的な検索を行うことができないことである．例えば，文全体，もしく

正した後のコード片である．3，4 行目がバッファオーバーフローを検出する部分である．

```
grep -nr S2TOS *

wconvert.c:2227:    req->datalen = requiredsize = S2TOS(p);
wconvert.c:2319: buf += HEADER_SIZE; Request.type2.context = S2TOS(buf);
wconvert.c:2331: buf += HEADER_SIZE; Request.type3.context = S2TOS(buf);
...
```

図 3.4: grep を用いた類似コード片検索

は単一の識別子をキーワードとして抽出し検索を行った場合，それぞれ以下の結果になると考えられる．

文全体をキーワードに指定した場合 文字列が完全に一致するコード片のみが出力され，識別子や構文に些細な表現上の差異があったコード片を検索結果に含めることはできない．

単一の識別子をキーワードに指定した場合 大量のコード片が提示されることが多く，検索結果の確認に大きな労力が必要となる可能性が高い．

図 3.2 を用いて説明すると，文全体（例えば，図 3.2(a) に含まれる Request.type7.yomilen = (short)S2TOS(buf)）をキーワードに指定し検索を行ったとしても，図 3.2(b) のいずれの部分も提示されない．また，コード片間に差異があることを考慮し単一の識別子（例えば buf）を指定すると，欠陥を含まないコード片が大量に提示されやすい．

grep は，キーワード検索に加えて正規表現を用いたパターン検索を行うことができる．しかし，grep の正規表現と対象ソフトウェアの両方を十分に理解した開発者でなければ，正規表現を用いたパターンを適切に指定することは難しい．

3.2.2 CCFinder を用いた類似コード片検索

CCFinder[39] 等のコードクローン検出ツールを利用して，修正を要するコード片とトークン列が等価なコード片を列挙する方法 [30] も考えられる．しかし，この方法では，構文がわずかに異なるなど些細な差異があるコード片を列挙することはできない．

例えば，図 3.2(a) の最終行の右辺は (short)S2TOS(buf) であるが，図 3.2(b) の最終行の右辺は (Ushort *)buf であり，字句解析によって得られるトークン列上に差異が存在する．そのため，CCFinder を用いて，一方のコード片とトークン列が等価なコード片を検出したとしても，もう一方のコード片を検出することはできない．

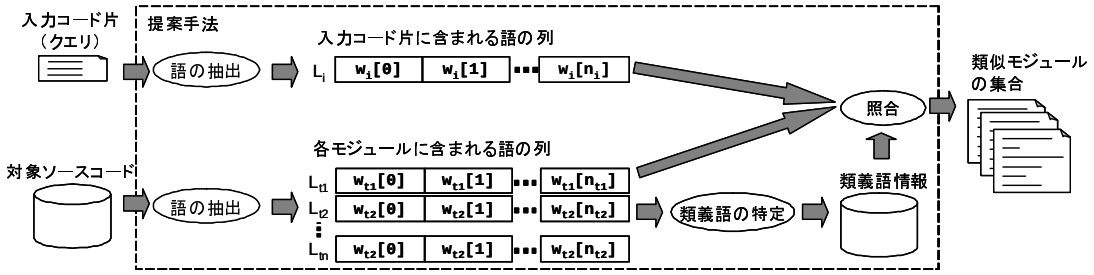


図 3.5: 提案手法の概要

3.3 提案手法

本稿では，入力コード片を与えると，識別子の類似性に基づいて対象ソースコードから類似関数を検索する手法を提案する．この手法を用いる開発者は，入力コード片を与えるのみで類似関数を検索できるため，キーワードを指定する必要がない．

提案手法の概要を図 3.5 に示す．提案手法は，以下に示す手順からなる．

手順 1（語の抽出） 対象ソースコードの各関数が含む**語**（識別子を分割・正規化した後の文字列）を抽出

手順 2（類義語の特定） 対象ソースコードが含む語の共起関係に基づいて類義語を特定

手順 3（入力コード片との照合） 入力コード片が含む全ての語と一致する，もしくは類義語である語を含む関数を類似関数として提示

以降，手順 1，2，3 について，それぞれ詳述する．

3.3.1 手順 1（語の抽出）

対象ソースコード中の各関数が含む識別子を抽出し，分割・正規化[2]を行う．本稿では，分割・正規化後の文字列を**語**と呼ぶ．その後，各関数が含む語の出現回数を表す行列を作成する．

図 3.6 は，語の出現回数を表す行列の例である．この行列は，関数 m_0，m_1，m_2 における語 w_a, w_b, w_c, w_d, w_e の出現回数を表している．以降，この例を用いて説明する．

[2]行った分割・正規化は，アンダースコアの位置での分割（例えば，add_host を add と host に分割）や接尾数字の削除（例えば，type7 の 7 を削除），キャメルケースに従った識別子を大文字から始まる語に分割（例えば，addHost を add と Host に分割）の全 3 種類である．これらの分割・正規化を行う理由は，分割後の語や接尾数字の削除が行われた語を含む関数を類似関数として提示することで，検索性能を向上させるためである．

$$
\begin{array}{c} \\ m_0 \\ m_1 \\ m_2 \end{array}
\begin{array}{c}
\begin{array}{ccccc} w_a & w_b & w_c & w_d & w_e \end{array} \\
\begin{pmatrix} 0 & 2 & 1 & 0 & 1 \\ 1 & 0 & 2 & 3 & 0 \\ 0 & 0 & 2 & 0 & 2 \end{pmatrix}
\end{array}
$$

図 3.6: 語の出現回数を表す行列の例

3.3.2 手順 2（類義語の特定）

ソースコード中に含まれる語を要素とする集合をクラスタリング（部分集合に分割）し，その結果 1 つのクラスタ（部分集合）に含まれた語を類義語とする[3]．本稿では，**ソースコードから類義語を特定する**とは，上述のようにソースコード中に含まれる語を要素とする集合をクラスタリングすることを言う．また，**2 つの語が類義語として特定される**とは，クラスタリングした結果それら語が同一のクラスタに含まれることを言う．

自然文書中に含まれる単語を対象にクラスタリングを行う基準 [73] を応用し，ソースコード中に含まれる語を対象にするクラスタリングとして以下の 2 つが考えられる．

(1) 語の共起性 語 w_x と語 w_y が頻繁に**共起**（同一関数など近い位置で出現）しているかどうかを基準とする．もし，頻繁に共起しているなら，同一クラスタに含める[4]．

(2) 共起している語の類似性 語 w_x と共起している語の集合が，語 w_y と共起している語の集合と類似しているかどうかを基準とする．もし，それら集合の類似性が高いなら，語 w_x と語 w_y を同一クラスタに含める[5]．

提案手法では，基準 (2) “共起している語の類似性” の計測モデルの 1 つである Dagan らのモデル [19] に基づくクラスタリングを行う．基準 (1) ではなく基準 (2) の特定法を採用した理由は，自然文書を対象とした実験 [19] において，言い換えを行っている単語などの類似した働きをする単語を特定しており，ソースコード中においても類似した概念名（例：入力データサイズと出力データサイズ）の全体または一部を表す語が存在するため，これらを類義語として特定できると考えたから

[3]語集合 S は独立した部分集合 $S_0, S_1, \ldots, S_n (S = \bigcup_{i=0}^{n} S_i$ かつ任意の $S_i, S_j (0 \leq i \leq n, 0 \leq j \leq n, i \neq j)$ について $S_i \cap S_j = \phi)$ に分割される．

[4]自然文書に含まれる類義語を特定する場合，この基準でクラスタリングを行うとクラスタ全体として 1 つの概念を表すことが多い．例えば，“doctor”，“nurse”，“hospital” が 1 つのクラスタになる [60]．

[5]自然文書に含まれる類義語を特定する場合，この基準でクラスタリングを行うと，類似した働きをする単語（例えば，言い換えを行っている単語）が同一のクラスタに含まれることが多い．例えば，“guy” と “kid” は共起している単語の類似性が高いと判定されている [19]．

$$
\begin{array}{c}
 \\ w_a \\ w_b \\ w_c \\ w_d \\ w_e
\end{array}
\begin{array}{c}
\begin{array}{ccccc} w_a & w_b & w_c & w_d & w_e \end{array} \\
\begin{pmatrix}
- & 0 & 1 & 1 & 0 \\
0 & - & 1 & 0 & 1 \\
1 & 1 & - & 1 & 2 \\
1 & 0 & 1 & - & 0 \\
0 & 1 & 2 & 0 & -
\end{pmatrix}
\end{array}
$$

(a) 図 3.6 を基に作成した共起回数を表す行列

$$
\begin{array}{c}
 \\ w_a \\ w_b
\end{array}
\begin{array}{c}
\begin{array}{ccccc} w_a & w_b & w_c & w_d & w_e \end{array} \\
\begin{pmatrix}
- & 0 & \mathbf{1} & \mathbf{1} & \mathbf{0} \\
0 & - & \mathbf{1} & \mathbf{0} & \mathbf{1}
\end{pmatrix}
\end{array}
$$

(b) 語 w_a と w_b の行（図 3.7(a) から抽出）

図 3.7: 類義語の特定に用いる行列

である．また，ソースコードにおいて，ある語と共起している語の集合と，その類義語（その語と類似した概念名を表す語）と共起する語の集合は，同一の関数名や同名の変数名を表す語を多く含み，類似しているのではないかと考えた．前述の通り，2 つの語の間で，共起する語の集合が類似していれば，基準 (2) の特定法でそれらを類義語と特定することができる．

なお，本稿では，**2 つの語が共起している**とは，それら語が同一関数内で出現していることを言う．また，**2 つの語が n 回共起している**とは，それら語が n 個の関数内で共起していることを言う．

Dagan らのモデルに基づいてクラスタリングを行う手順は，以下の (A) から (C) である．

(A) 語の共起回数を表す行列を作成 3.3.1 節の**手順 1** で作成した語の出現回数を表す行列（図 3.6）を基に，語の共起回数を表す**共起回数を表す行列**を作成する．共起回数を表す行列は，語の数を N とすると，$N \times N$ の対称行列で表される．共起回数を表す行列の要素 (w_x, w_y) は，語 w_x と語 w_y の共起（両者が 1 回以上出現）する関数の数を表す．なお，対角成分は用いないため，記号 “–” を置く．図 3.7(a) は，図 3.6 を基に作成した 5×5 の共起回数を表す行列である．この共起回数を表す行列では，語 w_c と語 w_e は 2 回共起していることを表している．

(B) 語の距離を算出 2 つの語の距離を，それらと共起している語の類似性に基づいて算出する．図 3.7(b) は，図 3.7(a) から語 w_a と w_b の行を抽出した行列である．太字で表す要素が，語 w_a および語 w_b の他の語（w_c，w_d，w_e）との共起回数を表している．語 w_a と語 w_c，w_d，w_e の共起回数の分布 $[1, 1, 0]$ と，

語 w_b と語 w_c，w_d，w_e の共起回数の分布 $[1, 0, 1]$ の距離を算出し，語 w_a と語 w_b の距離とする．2 つの分布の距離は，確率分布間の差異を表す尺度である Kullback-Leibler divergence[44] や Jensen-Shannon divergence[49] を用いて算出することができる．提案手法では，対称性を持つ Jensen-Shannon divergence を用いる．対称性のある Jensen-Shannon divergence を用いる理由は，単語間の距離に対称性を持たせることで，自然言語と同様に類義語であるという関係に対称性を持たせるためである[6]．また，提案手法を用いる開発者が，検索結果を理解するためには語間の関係に対称性がある方が容易に理解できると考えられる．例えば，語間の関係に対称性があるなら，単語 A が単語 B の類義語であることを提示するだけで，開発者は単語 B が単語 A の類義語であることがわかるが，対称性がない場合，開発者は単語 B が単語 A の類義語であることを確認する必要がある．

(C) **クラスタリング** (B) で求めた距離に基づいて，語のクラスタリングを行う．クラスタリングには様々な方法があるが，提案手法では，全ての語が独立したクラスタという状態から始めて，最も類似度の高いクラスタから順次結合していく方式を採用する．この方式では，クラスタ数が 1 になるか，もしくは任意のクラスタ間の距離が閾値以上になるまで結合を繰り返す．クラスタ間の類似度は**群平均法（平均距離法）**[68][70] を用いて求める．群平均法は，2 つのクラスタに属する要素間の距離の平均を求め，それらクラスタ間の類似度とする方法である．

3.3.3 手順 3（入力コード片との照合）

入力されたコード片と対象ソースコード中の各関数を照合することで，類似関数を提示する．本節では，提案手法が検出する語の対応関係および類似関数について述べる．

語の対応関係 2 つの語 W_x, W_y が与えられたとき，語 W_y が語 W_x と一致する，もしくは語 W_y が語 W_x の類義語であるなら，語 W_x は語 W_y と対応関係を持つとする．

類似関数 入力されたコード片 CF_i が含む n 個の語からなる列 $L_i = \left[Wi_0, Wi_1, \ldots, Wi_{n-1}, Wi_n\right]$ と比較対象の関数 M_t が含む m 個の語からなる列 $L_t = \left[It_0, It_1, \ldots, It_{m-1}, It_m\right]$ が与えられたとき，列 L_i が含む n 個の語 $Wi_0, Wi_1, \ldots, Wi_{n-1}, Wi_n$ のそれぞれと対応関係を持つ語がすべて列 L_t 中に存在するなら，関数 M_t をコード片 CF_i の類似関数とする．

[6] 自然言語において，上位語，下位語のような詳細な単語間の関係には対称性はないが，類義語であるかどうかという関係は一般的に対称性が存在する．

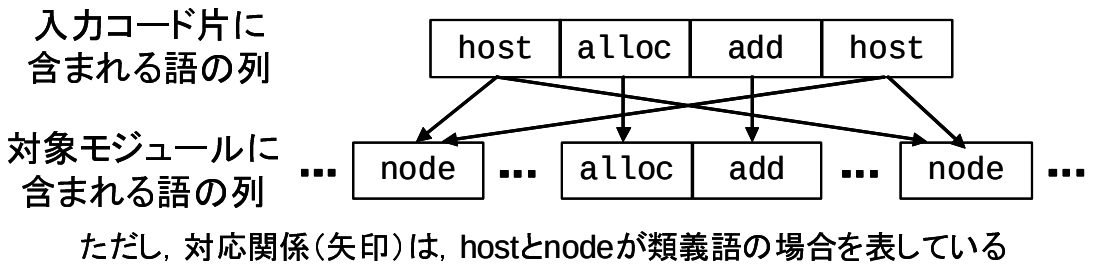


図 3.8: 入力コード片と対象関数の照合

図 3.8 は，入力コード片の語の列 $[host, alloc, add, host]$ と対象関数の語の列 $[node, \ldots, alloc, add, \ldots, node]$ の照合を表している．図中の矢印は，語の対応関係を表している．語 host と語 node が類義語として特定されている場合，入力コード片中の全ての語が対象関数中のいずれかの語と対応関係を持つ．よって，対象関数は類似関数として検出される．

3.4 適用実験

提案手法の有効性を確認するため，提案手法を実装し適用実験を行った．適用実験の目的は，保守対象のソースコード中に欠陥が見つかった際に，同種の欠陥を含む関数を探す作業に対して有効な支援ができているか確認することである．このために，提案手法に**欠陥コード片**（欠陥を含むコード片）を入力コード片として与え，同種の欠陥を含む関数を検索する実験を行った．加えて，grep[26] やコードクローン検出ツール CCFinder[39] についても同様の実験を行った．

適用対象には，オープンソースソフトウェア “かんな”[71] のバージョン 3.6 と我々の研究グループで開発している SPARS-J[74] を選んだ．適用対象のソフトウェアの構成を表 3.1 に示す．なお，これらソフトウェアのソースコードは，C 言語で開発されている．かんなは，クライアント・サーバ型の日本語入力システムである．かんなのバージョン 3.6 において，サーバ機能を実装した server ディレクトリ以下に 19 個のバッファーオーバーフローエラー（図 3.2，図 3.11）が含まれていた．それら 19 個の欠陥は，18 関数に含まれていた．適用実験では，サーバ機能に含まれるこれら欠陥を探す作業を支援できるか確認するため，server ディレクトリ以下の*.c ファイルを対象とした．SPARS-J は，ソフトウェア部品検索システムである．SPARS-J のあるバージョンにおいて，計 75 個の型キャスト忘れ[7]がシステム全体に渡って存在した．それら 75 個の欠陥は，50 関数（関数）に含まれていた．適用実験では，SPARS-J 全体に含まれるこれら欠陥を探す作業を支援できるか確

[7]ソフトウェア部品を登録するデータベースのデータ表現に合わせるための型キャストが欠如していた．

認するため，全ての*.c ファイルを対象とした．

3.4.1 提案手法の適用実験

本節で述べる実験の目的は，以下の 2 つである．

目的 1 語のクラスタリング（3.3.2 節参照）で用いる閾値を変化させることで，類義語として扱う語を増やしたときの検索結果の変化を確認する

目的 2 入力コード片を変化させたときの検索結果の変化を確認する

目的 1 に用いる閾値 th_r は，0 以上 1 以下の値をとる r が与えられたときに，以下に手順で求めることができる．

1. 対象ソフトウェアに含まれる任意の語の距離を求め，その最大値を d_{max} とする．
2. 最大値と r の積を th_r とする．

なお，実験では，6 つの閾値 $th_{0.0}$，$th_{0.1}, \ldots, th_{0.5}$ を設定した．

目的 2 のために，対象ソースコード中の各欠陥を含むクエリを作成し検索を行った．かんなの場合 19 個，SPARS-J の場合 75 個の入力コード片を作成し，各入力コード片を 1 つずつ与え検索を行った．かんなに与えた各入力コード片は，バッファからの読み込み処理（図 3.2 参照）を行っている連続する行から構成され，SPARS-J の場合は，型キャスト忘れを含む 1 行もしくは連続する行から構成される．

検索結果を評価するために，検索システムの性能評価に用いられる適合率，再現率，F 値を計測した [69]．本稿で用いる適合率 $Precision$，再現率 $Recall$，F 値 F は，D を欠陥関数の集合，R を検索された関数の集合とすると，以下のように表される．

$$Precision = \frac{|D \cap R|}{|R|} \tag{3.1}$$

$$Recall = \frac{|D \cap R|}{|D|} \tag{3.2}$$

$$F = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{3.3}$$

表 3.1: 適用対象のソフトウェア

名前	総行数	関数の数	欠陥関数の数	欠陥数
かんな	7,613	203	18	19
SPARS-J	35,744	859	50	75

適合率は検索結果の正確性（検索ノイズの少なさ）を表しており，再現率は検索結果の完全性（検索漏れの少なさ）を表す．欠陥関数を探す作業を支援する手法には，検索漏れの少なさを表す再現率が高いことに加え，適合率が高いことが求められる．その理由は，ソフトウェア保守の現場では，全関数を網羅的に検査するための資源を獲得できるとは限らないため，再現率が高いだけでなく適合率が高いことも求められるからである．F 値は，適合率と再現率の調和平均である．F 値は，検索性能を 1 つのスカラ値として表現でき，また適合率あるいは再現率の一方だけが高い極端な場合も正当に評価することができる [72].

図 3.9，図 3.10 は，閾値を $th_{0.0}$ から $th_{0.5}$ まで変化させたときの適合率・再現率の変化を表すグラフである．それぞれのグラフにおいて，閾値ごとに 1 入力コード片あたりの平均値と，全クエリ中の最大値・最小値をプロットしている．図 3.9，図 3.10 から，閾値 th_r を増加させると，適合率は低下し，再現率は上昇する傾向にあることがわかる．

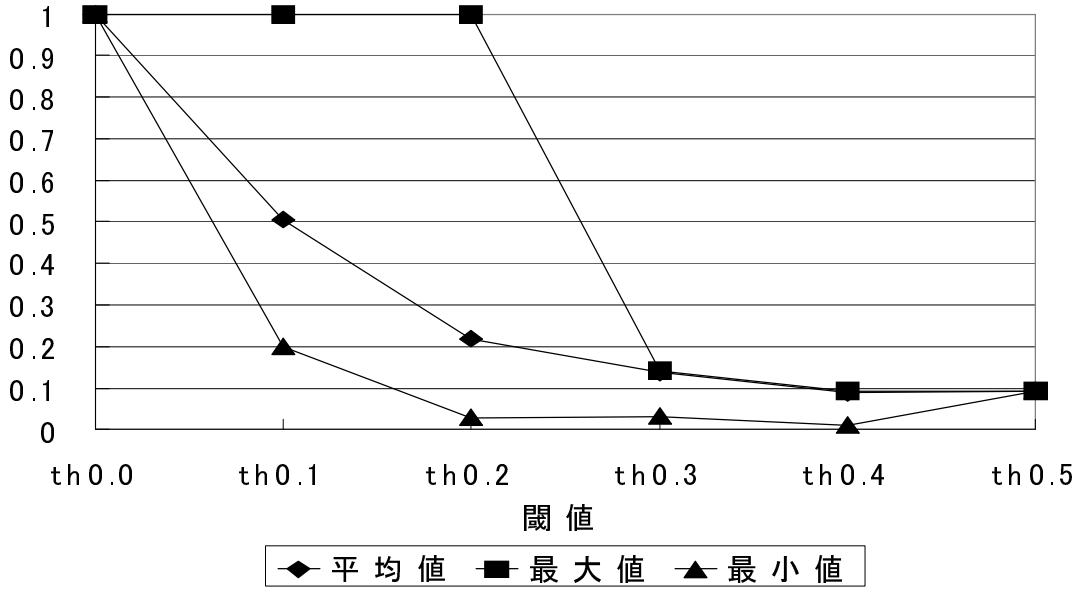
閾値が $th_{0.0}$ の場合は検出漏れが多かった．また，かんなの場合は閾値を $th_{0.4}$ や $th_{0.5}$ に設定すると，対象ソースコードに含まれる全てもしくはほとんどの関数を検出し，SPARS-J の場合は $th_{0.3}$ や $th_{0.4}$，$th_{0.5}$ に設定すると同様に全てもしくはほとんどの関数を検出した．このような検索結果を提示しても欠陥関数の検査作業を支援することはできない．よって，かんなや SPARS-J が対象の場合に有効な支援をできる可能性がある閾値は $th_{0.1}$ および $th_{0.2}$ であると言える．

表 3.3 は，Canna を対象として，閾値を $th_{0.1}$ もしくは $th_{0.2}$ に設定したときのクラスタリング結果を表している．クラスタ数が多く，また膨大な数の単語を含むクラスタが多く存在したため，一部クラスタに含まれた一部の単語のみを抜粋している．閾値を $th_{0.1}$ から $th_{0.2}$ に変化させると，クラスタ 1-A と 1-F が結合されクラスタ 2-A になり，クラスタ 1-B，1-C，1-D，1-E が結合されクラスタ 2-B になった．クラスタ 1-G についてはいずれのクラスタとも結合しなかった（表 3.3(b) のクラスタ 2-C）．

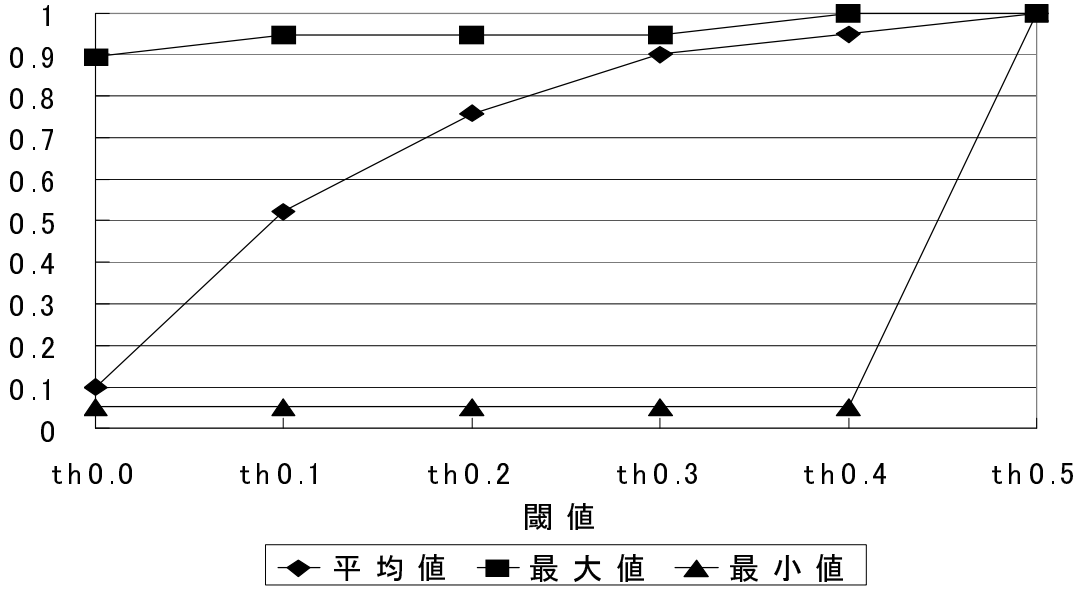
語のクラスタリング結果が検索結果にどのように影響したかを考察するため，入力コード片を 3 つ選び，閾値を $th_{0.1}$，$th_{0.2}$ に設定したときの検索結果を示す．なお，これら閾値を選んだ理由は，提案手法が有効に機能する入力コード片と機能しない低い入力コード片の両者が存在するため，提案手法の有効性が高い場合と低い場合を比較・議論しやすいからである（他の閾値を選ぶと，提案手法が有効に機能する入力コード片が非常に少ない）．検索結果に関して，以下に示す特徴がある 3 つのコード片 CF_A，CF_B，CF_C（図 3.11）を選んだ．

コード片 CF_A 閾値を $th_{0.1}$ から $th_{0.2}$ に変化させると，適合率が大きく低下し，かつ再現率が上昇したコード片

コード片 CF_B 閾値を $th_{0.1}$ から $th_{0.2}$ に変化させると，適合率はほとんど変化しなかったが，再現率が上昇したコード片

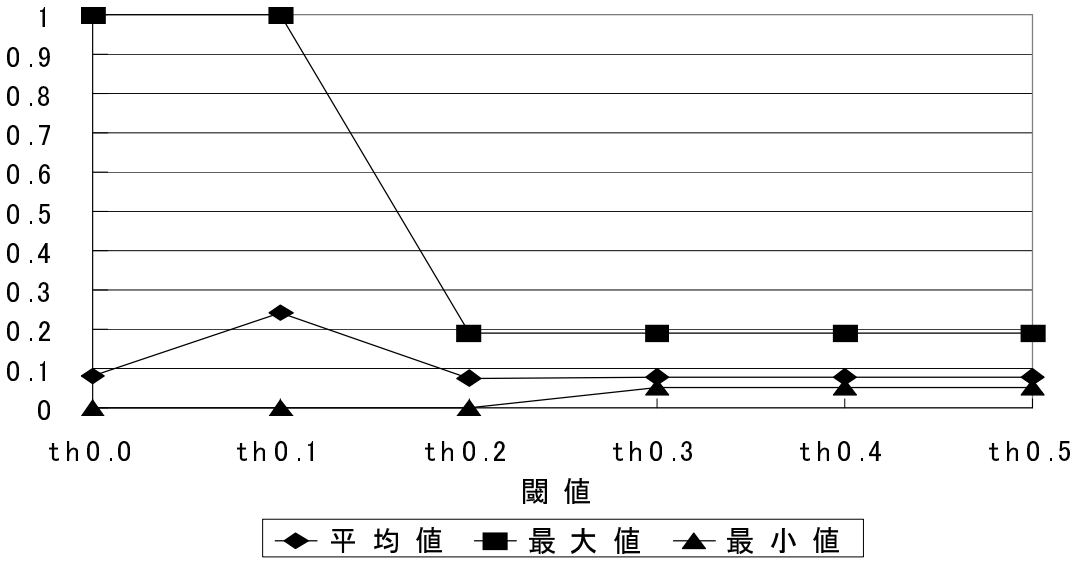


(a) かんなの適合率

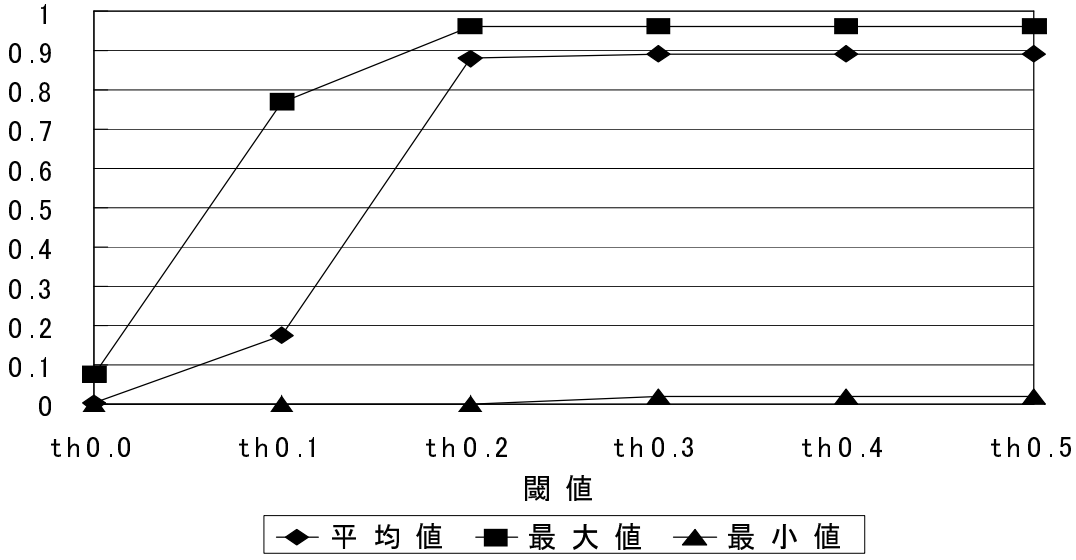


(b) かんなの再現率

図 3.9: かんなの実験結果



(a) SPARS-J の適合率



(b) SPARS-J の再現率

図 3.10: SPARS-J の実験結果

```
buf += HEADER_SIZE; Request.type7.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.number = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.yomilen = (short)S2TOS(buf);
```

(a) コード片 CF_A

```
buf += SIZEOFINT; Request.type10.kouho = (short *)buf; /* short? */
for (i = 0; i < Request.type10.number; i++) {
  Request.type10.kouho[i] = S2TOS(buf); buf += SIZEOFSHORT;
  ir_debug(Dmsg(10, "req->kouho =%d\n", Request.type10.kouho[i]));
}
```

(b) コード片 CF_B

```
buf += HEADER_SIZE; Request.type13.context = S2TOS(buf);
len = SIZEOFSHORT ;
buf += len;
Request.type13.dicname = (char *)buf;
len = strlen( (char *)buf ) + 1;
buf += len;
Request.type13.yomi = (Ushort *)buf;
len = ((int)Request.type13.datalen - len - SIZEOFSHORT * 4)
/ SIZEOFSHORT;
for( data = Request.type13.yomi, i = 0; i < len; i++, data++)
  *data = ntohs( *data );
buf += (ushortstrlen((Ushort *)buf) + 1) * SIZEOFSHORT;
Request.type13.yomilen = S2TOS(buf);
buf += SIZEOFSHORT; Request.type13.kouhosize = S2TOS(buf);
buf += SIZEOFSHORT; Request.type13.hinshisize = S2TOS(buf);
```

(c) コード片 CF_C

図 3.11: コード片 CF_A，CF_B，CF_C

コード片 CF_C 閾値を $th_{0.1}$ から $th_{0.2}$ に変化させると，適合率は低下したが，再現率は変化しなかったコード片

表 3.2(a) に，クラスタリングの閾値を $th_{0.1}$ もしくは $th_{0.2}$ に設定した提案手法に対して，コード片 CF_A，CF_B，CF_C を入力したときの適合率・再現率・F 値を示す．最も F 値が良かった場合は，$th_{0.1}$ に設定したときのコード片 CF_A であった．次に良かった場合は，$th_{0.2}$ に設定したときのコード片 CF_A と CF_B であり，同じ値（0.31）であった（なお，これらは，全く同じ検索結果であった）．以降，$th_{0.1}$ に設定したときのコード片 CF_B，$th_{0.1}$ に設定したときのコード片 CF_C，$th_{0.2}$ に

表 3.2: 各コード片の検索結果

(a) 提案手法

	提案手法 ($th_r = 0.1$)			提案手法 ($th_r = 0.2$)		
	適合率	再現率	F 値	適合率	再現率	F 値
CF_A	0.50	0.72	0.59	0.18	1.00	0.31
CF_B	0.20	0.33	0.25	0.18	1.00	0.31
CF_C	1.00	0.06	0.11	0.33	0.06	0.10

(b) grep，CCFinder

	grep (キーワード: buf)			CCFinder		
	適合率	再現率	F 値	適合率	再現率	F 値
CF_A	0.19	1.00	0.32	1.00	0.06	0.11
CF_B	0.19	1.00	0.32	1.00	0.06	0.11
CF_C	0.19	1.00	0.32	1.00	0.06	0.11

設定したときのコード片 CF_C の順であった．

表 3.4 に，各コード片に含まれる語が属したクラスタを示す．まず，閾値が $th_{0.1}$ の場合について述べる．コード片 CF_A と CF_B の検索結果は表 3.2(a) で示したとおり大きく異なっていたが，対応するクラスタは一部のみ（CF_A にはクラスタ 1-C が対応していたが，CF_B にはクラスタ 1-E が対応）が異なっていた．また，検索結果が悪かったコード片 CF_C には，他のコード片と比べて多くのクラスタが対応した．

閾値を $th_{0.2}$ に設定すると，コード片 CF_A と CF_B には同じクラスタ群（クラスタ 2-A および 2-B）が対応したため，全く同じ検索結果が得られた．閾値が $th_{0.1}$ の場合に異なるクラスタであったクラスタ 1-C と 1-E は，クラスタ 2-B に包含されていた．コード片 CF_C については，クラスタ 2-A および 2-B に加えて，クラスタ 1-G と同一の語からなるクラスタ 2-C も対応した．その結果，閾値を $th_{0.1}$ に設定したときと同様に，コード片 CF_C を含む関数のみが検索結果に含まれた．

3.4.2 grep や CCFinder との比較実験

提案手法と grep，CCFinder の有効性を比較するために実験を行った．grep を用いた実験では，コード片 CF_A，CF_B，CF_C から抽出したキーワードを grep に与えることで，キーワードを含む関数を検出し，類似関数の検索を行った．また，CCFinder を用いた実験では，各コード片とコードクローンになっているコード片を含む関数を検出することで，類似関数の検索を行った．

表 3.2(b) は，コード片 CF_A，CF_B，CF_C を入力して，grep や CCFinder を用い

表 3.3: 各クラスタに属していた語（抜粋）

(a) $th_r = 0.1$

	語数	コード片 CF_A，CF_B，CF_C のいずれかに含まれた語
クラスタ 1-A	7	buf, HEADER, S2TOS, SIZE, SIZEOFINT SIZEOFSHORT (6 語)
クラスタ 1-B	43	context, data, debug, dicname, ir, kouho, number, type, Request (9 語)
クラスタ 1-C	5	i, len (2 語)
クラスタ 1-D	2	strlen (1 語)
クラスタ 1-E	12	datalen, ushortstrlen, yomi, yomilen (4 語)
クラスタ 1-F	1	ntohs (1 語)
クラスタ 1-G	4	hinshisize, kouhosize (2 語)

(b) $th_r = 0.2$

	語数	コード片 CF_A，CF_B，CF_C のいずれかに含まれた語
クラスタ 2-A	77	buf, ntohs, HEADER, S2TOS, SIZE, SIZEOFINT, SIZEOFSHORT (7 語)
クラスタ 2-B	126	context, data, datalen, debug, dicname, i, ir, kouho, len, number, strlen, type, ushortstrlen, yomi, yomilen, Dmsg, Request, Ushort (18 語)
クラスタ 2-C	4	hinshisize, kouhosize (2 語)

表 3.4: コード片に含まれる語に属するクラスタ

(a) $th_r = 0.1$

	対応するクラスタ
コード片 CF_A	クラスタ 1-A，クラスタ 1-B，クラスタ 1-E
コード片 CF_B	クラスタ 1-A，クラスタ 1-B，クラスタ 1-C
コード片 CF_C	クラスタ 1-A，クラスタ 1-B，クラスタ 1-C，クラスタ 1-D，クラスタ 1-E，クラスタ 1-F，クラスタ 1-G

(b) $th_r = 0.2$

	対応するクラスタ
コード片 CF_A	クラスタ 2-A，クラスタ 2-B
コード片 CF_B	クラスタ 2-A，クラスタ 2-B
コード片 CF_C	クラスタ 2-A，クラスタ 2-B，クラスタ 2-C

た実験を行った結果である．grep の結果については，識別子 buf をキーワードとして指定した場合を代表例として掲載している．識別子 buf を代表例として選んだ理由は，対象とする欠陥である buf が指すバッファのオーバーフローに最も関係が深いため，一般の開発者が grep のキーワードに指定する可能性が最も高いと考えられるからである（他のキーワードを指定した場合については，表 3.5 に掲載）．識別子 buf をキーワードとして指定した場合の grep は，再現率は 1.00 であり非常に高かったが，適合率は 0.19 であり低かった．CCFinder については，各コード片を含む関数しか検索結果に含まれなかった．F 値を基準に各検索結果の比較を行うと，提案手法の閾値を $th_{0.1}$ に設定したときのコード片 CF_A が最高値 (0.59) であった．次いで，grep と閾値を $th_{0.2}$ に設定したときのコード片 CF_A，CF_B がほぼ同じ値 (0.31〜0.32) で並び，その次は閾値を $th_{0.1}$ に設定したときのコード片 CF_B であった．その他の場合は，各コード片を含む関数しか検索結果に含まれず，F 値も低い値 (0.10〜0.11) であった．

表 3.5 は，buf およびそれ以外のキーワードをコード片 CF_A，CF_B，CF_C から抽出し，grep に与えたときの検索結果を表している．いずれのコード片から抽出したキーワードにおいても，キーワードにより結果が大きくばらついた．

3.5 考察

3.5.1 語のクラスタリングに用いる閾値について

閾値の増加に伴い再現率が増加していた．このことから，類義語の範囲を広げることで，より多くのコード片を類似関数として検索結果に含めることができたと考えられる．

語のクラスタリングに用いる閾値が $th_{0.0}$ の場合は検出漏れが多かった．また，かんなの場合は閾値を $th_{0.4}$ や $th_{0.5}$ に設定すると，対象ソースコードに含まれる全てもしくはほとんどの関数を検出し，SPARS-J の場合は $th_{0.3}$ や $th_{0.4}$，$th_{0.5}$ に設定すると同様に全てもしくはほとんどの関数を検出した．このような検索結果を提示しても欠陥関数の検査作業を支援することはできない．よって，かんなや SPARS-J が対象の場合に有効な支援をできる可能性がある閾値は $th_{0.1}$ および $th_{0.2}$ であると言える．

以上のことから，かんなや SPARS-J を対象とした場合，再現率を重視するならば閾値を $th_{0.2}$ に設定し，逆に適合率を重視するならば閾値を $th_{0.1}$ に設定すると良いと考えられる．3.4.1 節で述べたように，ソフトウェア保守の現場では全ての関数を網羅的に検査するための資源を獲得できるとは限らないため，適合率を重視し，閾値を $th_{0.1}$ に設定する状況は十分に考えられる．

対象ソフトウェアによって有効な閾値が変化する可能性があるため，他のソフトウェアを対象とした実験を通して，多くのソフトウェアに有効な閾値の決定方法を考案する必要がある．現状では，対象ソースコードに依存しない一般に有効

表 3.5: grep による検索の結果

(a) コード片 CF_A，CF_B，CF_C の全てに含まれる部分をキーワードとして与えた場合

キーワード	検出行数	適合率	再現率	F 値
buf	557	0.19	1.00	0.31
HEADER_SIZE	46	0.67	1.00	0.80
Request	323	0.17	1.00	0.30
context	211	0.17	0.94	0.29
S2TOS	40	0.94	0.94	0.94
SIZEOFSHORT	87	0.53	0.89	0.67
buf += HEADER_SIZE	18	1.00	1.00	1.00
buf += SIZEOFSHORT	30	1.00	0.89	0.94

(b) コード片 CF_A にのみ含まれる部分をキーワードして与えた場合

キーワード	検出行数	適合率	再現率	F 値
type7	11	0.33	0.06	0.10

(c) コード片 CF_B にのみ含まれる部分をキーワードして与えた場合

キーワード	検出行数	適合率	再現率	F 値
type10	17	0.25	0.06	0.10
kouho	48	0.25	0.06	0.09
ir_debug	260	0.18	1.00	0.31
Dmsg	266	0.18	1.00	0.30

(d) コード片 CF_C にのみ含まれる部分をキーワードして与えた場合

キーワード	検出行数	適合率	再現率	F 値
type13	21	0.50	0.06	0.10
len	398	0.10	0.56	0.17
dicname	148	0.09	0.22	0.13
strlen	70	0.07	0.17	0.10
yomi	129	0.19	0.28	0.23

な閾値の決定法を実現できていない．よって，有効な閾値を求めるために，何度か検索を繰り返さなければならない場合が多いと考えられる．

多くの閾値において，最大値・平均値・最小値の大きな差があった．提案手法の有効性を高めるためには，これら値の差を縮める必要があると考えられる．

3.5.2 語のクラスタリングについて

クラスタ 1-A には，識別子 buf が表すバッファへのポインタに加算する定数の名前（例：HEADER_SIZE）や，buf の特定ビット列を取得するためのマクロ名が含まれていた．buf と他の語が同一関数中で共起することが多かったため，これらの語の共起回数の分布が類似し，同一クラスタに含まれていたと考えられる．

クラスタ 1-B，1-E には，構造体の要素を指定する語が含まれていた．構造体の各要素を指定するためには，これらの語を同時に使用する必要があるため，共起回数の分布が類似し，クラスタ 1-B もしくはクラスタ 1-E に含まれたと考えられる．なお，閾値を $th_{0.2}$ に設定すると，クラスタ 1-B とクラスタ 1-E は結合され，1 つのクラスタになった．

クラスタ 1-C は，C 言語を用いた開発において良く用いられる語を含んでいた．これらの語は，比較的多くの関数に共通して用いられたため，共起回数の分布が類似し，同一クラスタに含まれた．しかし，特定のプログラミング言語を用いた開発において，良く用いられる語（例えば，i と len）が，類似した役割を担っている場合は少ないと考えられる．よって，このような語をフィルタリングする方法を検討する必要があると考えられる．

クラスタの中には，含まれる語の数が少ないものがあった（クラスタ 1-C，1-D，1-F，1-G，2-C）．よって，全てのクラスタを含むコード片だけでなく，1 つ以上のクラスタを含むコード片を提示できる方法に改善する必要があると考えられる．この方法を採用すると，提示する関数の数が多くなりすぎる可能性があるため，対応関係の数に基づく順位付けを提示する関数に対して行い，上位の関数から順に提示する必要があると考えられる．

3.5.3 grep との比較について

F 値を基準すると，閾値を $th_{0.1}$ に設定した提案手法にコード片 CF_A を与えた場合が最も高い値であり，buf をキーワードとして grep に与えた場合よりも高い値であった．次いで，閾値を $th_{0.1}$ に設定した提案手法にコード片 CF_A もしくは CF_B を与えた場合の検索結果が高い F 値であり，buf をキーワードとして grep に与えた場合とほぼ同じ値であった．その他の場合は，grep の方が高い値であった．

提案手法が提示した検索結果の F 値が，buf をキーワードとして grep に与えたときよりも低い値になった場合があった要因として，以下が考えられる．

(1) **多くの関数に出現する語からなるクラスタ** コード片 CF_B は語 i を，コード片 CF_C は語 i，len を含んでおり，これらコード片は，いずれの閾値の場合もコード片 CF_A より F 値が低かった．また，閾値を $th_{0.1}$ に設定したとき，語 i，len は両者ともクラスタ 1-C に属した．3.5.2 節で述べたように，クラスタ 1-C に含まれるような，C 言語を用いた開発において良く用いられる語が類似した役割を担っていることは少ないと考えられる．

(2) **語数の少ないクラスタ** 再現率が低かったコード片 CF_C に属する語は，他のコード片に属する語に比べて，語数の少ないクラスタ（クラスタ 1-C，1-D，1-F，1-G，2-C）に含まれていることが多かった．入力コード片が語数が少ないクラスタをいくつか含むと，提案手法が提示する関数の数が大きく減り，再現率が下がると考えられる．

これら要因の影響を受けなかったと考えられる閾値を $th_{0.1}$ に設定したコード片 CF_A，および閾値を $th_{0.2}$ に設定したコード片 CF_A，CF_B については，提案手法の結果は grep より高い F 値であるか，もしくはほぼ同じ F 値であった．これら要因を受けない場合，提案手法は grep よりも高い，もしくは同等の有効性を持つ可能性があると考えられる．これら要因の影響を低減させるためには，3.5.2 節で述べたように，多くの関数に出現する語のフィルタリングする手法や，全てのクラスタを含むコード片だけでなく 1 つ以上のクラスタを含むコード片を提示できる手法に改善する必要があると考えられる．

コード片 CF_B については，閾値を $th_{0.1}$ に設定した場合は grep と比べて再現率・F 値が低かったが，閾値を $th_{0.2}$ に変化させると，再現率・F 値が上昇し，grep と同等の検索性能であった．これは，閾値を $th_{0.1}$ に設定したときに存在した少数の語からなるクラスタ 1-C（5 語）が，閾値を $th_{0.2}$ に変化させると他の複数のクラスタと結合し，多数の語からなるクラスタ 2-B（126 語）が構成され，多くの関数を提示するようになったためと考えられる．一方，コード片 CF_A については，閾値を $th_{0.1}$ に設定した場合に，既に grep と比べて適合率・F 値が高かった．そのため，閾値を $th_{0.2}$ に変化させると，クラスタ 1-E がクラスタ 1-C など他のクラスタ結合し語数の多いクラスタ 2-B を構成し，適合率が下がったと考えられる．これらのことから，以下のことが言える．

- 閾値を上昇させると，コード片間の検索結果の差異が小さくなる．
- 語数の少ないクラスタが原因で再現率が低い場合に閾値を上昇させると，そのクラスタと他のクラスタが結合し，適合率は下がるが再現率が大幅に上昇する可能性がある．
- 既に検索結果が良い場合に閾値を上昇させると，入力コード片に対応するクラスタが他のクラスタと結合し，再現率は上昇するものの適合率が大幅に下がる可能性がある．

grep の実験については，主に buf をキーワードとして与えた場合を取り上げたが，表 3.5 に示すとおり，キーワードにより結果が大きくばらついている．キーワードを適切に指定できる開発者であれば grep の有効性は高いと言えるが，逆にキーワードを適切に指定できない開発者あれば，提案手法の方が有効性が高い場合があると考えられる．

3.5.2 節で述べたように，提案手法は対象ソースコードに依存しない一般に有効な閾値の決定法を実現できていないため，有効な閾値を決めるための作業量が必要となる．grep は，コード片からキーワードを抽出する必要がある代わりに，閾値を設定する必要がないため，全体的な作業量では grep の方が少ない場合もあり得ると考えられる．これらのことから，提案手法と grep 等の各ツールについて全体の作業量の比較実験を行う必要があると考えられる．

行単位で検出を行う grep の方がより詳細に欠陥を含むコード片の位置を提示できると考えられるが，grep を用いて検出した各行を確認することで検査を行うことが難しい場合もある．具体的には，複数行からなるコード片や小規模の関数全体が 1 つの欠陥を表している場合に grep を用いると，これらコード片中や関数中に含まれる数行を提示することが多く，提示された各行を確認しただけでは欠陥の有無を確認することは難しい．

3.5.4 CCFinder との比較について

本稿の適用実験では，トークン列から連続して一致する部分列を検出する CCFinder を用いて，入力コード片のトークン列と連続して一致するトークン列を含む関数を検出した．しかし，（入力コード片を含む関数を除く）全ての欠陥関数は連続して一致するトークン列を持たなかったため，検出されなかった．よって，コード片 CF_A，CF_B，CF_C を入力コード片として与える場合は，トークン列でなく識別子の類似性に基づいて類似関数を提示する提案手法の方が有効であると言える．

本来 CCFinder は，コードクローン検出ツールとして開発されているため，類似コード片検索を行う多くの場合において有効性は低いと考えられる．しかし，対象とする類似コード片によっては，CCFinder のように等価なトークン列を検索した方が有効な場合もあると考えられるため，提案手法と CCFinder を使い分ける必要があると考えられる．提案手法と CCFinder の比較実験を更に行うことで，それぞれが有効な類似コード片の性質を明らかにする必要がある．

3.6 関連研究

これまでに様々なコードクローン検出法が提案されており，その中には CCFinder 等のトークン列の等価性に基づく検出法だけでなく，抽象構文木やプログラム依存グラフの等価性に基づく検出法も提案されている [13, 37, 42, 24]．抽象構文木

の等価性に基づく検出法の中には，抽象構文木が完全一致しなくても，主要な構文要素が一致していれば，コードクローンとして検出する手法も提案されている [37]．また，プログラム依存グラフの等価性に基づく手法 [42] は，構文が等価でなくてもプログラム依存グラフが等価であれば，コードクローンとして検出する．提案手法は，識別子の類似性に基づいて類似コード片を検索するため，構文上やプログラム依存グラフが異なるコード片であっても検索結果に含めることができる．対象とする類似コード片によっては，上述のコードクローン検出法を用いて類似コードを検索した方が有効な場合もあると考えられる．よって，提案手法とこれらコードクローン検出法の比較実験を行うことで，各手法が有効な類似コード片の特徴を明らかにする必要がある．

Splint[21] 等の lint 系のツールや FindBugs[31] のように，ソースコード中から検査すべき部分を検出するツールが数多く開発されている．これらツールの多くには，初期化されていない変数への参照や一度も参照されない変数を検出するアルゴリズムが実装されている．しかし，ドメインやアプリケーションに特化した欠陥を検出する機能を追加するためには，ツールを拡張する必要がある．提案手法は入力コード片を必要とする代わりに，対象ソースコードやそのドメインに特化した欠陥であっても，欠陥を含むコード片を抽出し入力コード片として与えるだけで類似コード片を検索することができるため，ドメインやアプリケーションに特化した欠陥を検出できる可能性がある．

Li らが提案する PR-Miner は，Frequent Itemset Mining アルゴリズムを用いて，変数名や関数名の出現パターンを特定し，パターンに違反する変数名や関数名の出現を欠陥候補として提示する [48]．PR-Miner によって，欠陥を含むコード片を検出し，類義語の特定を行う提案手法を用いてそのコード片の類似コード片を検索すると，新たな欠陥を検出できる可能性がある．また，Li らは，コピーアンドペーストにより作り込まれたコード片に対して一貫した識別子の修正が行われているか判定し，一貫した修正が行われていないコード片を欠陥候補として検出する手法を提案している [47]．この手法についても，PR-Miner と同様に，欠陥を含むコード片を検出し提案手法に入力すると新たな欠陥を検出できる可能性がある．

適用実験において，提案手法を用いて欠陥があるコード片の類似関数を検索することで，欠陥関数を検出した．しかし，提案手法が行うことは飽くまで類似コード片検索であるため，欠陥を含む検出が目的の場合は上述の欠陥検出ツールの出力結果と照合し，絞り込みや順位付け（例えば，複数のツールが欠陥を検出したコード片を上位に順位付け）を行う必要があると考えられる．

3.7 結論

本稿では，入力として与えたコード片に類似したコード片を検索する手法を提案した．提案手法は，完全に一致する識別子を含むコード片を検出するだけでなく，類義語を含むコード片を類似コード片として検出することができる．提案手法

を実装し，複数の類似した欠陥を持つソースコードに対して適用したところ，有効な検索を行うためには類義語の特定に用いる閾値を適切に決める必要があることがわかった．更に，提案手法と grep，コードクローン検出ツール CCFinder にそれぞれ同じ入力コード片を与え類似コード片検索を行うことで，結果の比較を行った．提案手法と grep を比較すると，提案手法が有効に働く入力コード片と grep が有効に働く入力コード片がそれぞれ存在した．一方，CCFinder は（入力コード片を含む関数を除く）欠陥関数を検出することはできなかったため，提案手法の方が有効な検索を行えたと言える．

今後，検索性能を向上するために，入力コード片と対象コード片の照合方法を改善（3.5.2 節参照）したいと考えている．更に，現状の提案手法は構文情報を用いていないため，構文情報を用いた検索を実現したいと考えている．具体的には，特定した類義語を含む部分の構文が一致しているコード片のみを提示する手法に改善する．有用性の調査として，他のソフトウェアに含まれる欠陥の検出や，同時に機能拡張する必要のある関数の検出といった他の用途に提案手法を適用する予定である．

第4章　むすび

4.1　まとめ

本論文では，リファクタリングや類似コード検索を行う上での，既存のコードクローン検出ツールの問題点に着目し，その解決を試みた．

まず，コードクローン間の依存関係に基づくリファクタリング支援手法では，既存のコードクローン検出ツールがクローンセット間の依存関係を検出しないために，クローンセットの集約が困難になる場合があるという問題点に取り組んだ．提案手法は，クローンセット中に存在するメソッド呼び出し関係，およびメソッドと変数の利用関係を解析することで，ソースコード中からチェーンドクローンセット（同時に集約を検討すべきクローンセットの集合）を検出することで，クローンセットの集約作業を支援する．更に，検出したチェーンドクローンセットを含むクラスからなる集合の継承関係から，適用可能なリファクタリングパターン（“Extract Method” や，“Pull Up Method”，“Extract Superclass”）を提示する．適用実験では，オープンソースソフトウェアのソースコードに含まれているチェーンドクローンセットの規模を調査した．その後，検出したそれらチェーンドクローンセットに対して，提案手法が提示するリファクタリングパターンを適用した．その結果，チェーンドクローンセットの規模がクローンセットと比べて大きいこと，およびチェーンドクローンセットのリファクタリングがクローンセット単位のリファクタリングと比べて容易であることを確認できた．

次に，類義語の特定に基づく類似コード検索法では，既存のコードクローン検出ツールがトークン列や構文上差異がある類似コード片を検出できないことが多いという問題点に取り組んだ．本手法は，クエリとしてコード片を与えると，識別子の類似性に基づいて対象ソースコードから類似関数（クエリとして与えられたコード片の類似コード片を含む関数）を検索する．具体的には，まず自然言語処理の分野で提案されている類義語特定法を用いて，語（識別子を分割・正規化した後の文字列）の類義語を特定する．次に，クエリとして与えられたコード片に含まれる全ての語について，同一もしくは類義語である語を含む関数を検出し，類似関数として提示する．提案手法を実装し，複数の類似した欠陥を持つソースコードに対して適用したところ，有効な検索を行うためには類義語の特定に用いる閾値を適切に決める必要があることが確認できた．更に，提案手法と grep，コードクローン検出ツール CCFinder にそれぞれ同じ入力コード片を与え類似コード片検索を行うことで，結果の比較を行った．提案手法と grep を比較すると，提案手

法が有効に働く入力コード片と grep が有効に働く入力コード片がそれぞれ存在することが確認できた．一方，CCFinder は（入力コード片を含む関数を除く）欠陥関数を検出することはできなかったため，提案手法の方が有効な検索を行えることが確認できた．

4.2 今後の研究方針

今後，本論文で述べた研究成果を応用し，ソフトウェア保守作業をより容易なものにしていきたいと考えている．

本論文で扱ったリファクタリング支援を目的とした研究分野において，リファクタリング間の依存関係が問題として指摘されている [18, 52]．リファクタリング間の依存関係とは，あるリファクタリングを行うためには，その前に他のリファクタリングを行う必要があるという関係のことを指す．開発者に対して，リファクタリングを行う前にリファクタリング間の依存関係を正確に提示することができれば，効果的なリファクタリング支援を行うことができると考えられる．本論文で提案したコードクローン間の依存関係に基づくリファクタリング支援手法は，コードクローンのリファクタリングのみを対象にリファクタリング間に依存関係を提示していると考えることができる．提案手法が行うメソッド呼び出し関係やフィールドの共有関係の検出を応用し，コードクローンを対象としたリファクタリング以外のリファクタリング間の依存関係を解析する手法を提案したいと考えている．

また，本論文で類義語の特定に基づく類似コード片検索手法を提案し，適用実験において欠陥を含む関数の検索に応用した．しかし，類似した機能が実装されたコード片の検索への適用や，欠陥報告をクエリとして同様の欠陥を含むコード片を検索することに適用するなど，提案手法には様々な適用分野が考えられる．今後，他のソフトウェアに含まれる欠陥への適用のみならず，様々な用途に提案手法を適用していきたいと考えている．

参考文献

[1] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact strring matching based on suffix arrays. In *Proc. of SPIRE 2002*, pp. 31–43, 2002.

[2] A. Aiken. A system for detecting software plagiarism (moss homepage). `http://www.cs.berkeley.edu/~aiken/moss.html`.

[3] ANTLR. http://www.antlr.org.

[4] B. S. Baker. A program for identifying duplicated code. In *Proc. of Computing Science and Statistics*, Vol. 6, pp. 49–57, 1992.

[5] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. of WCRE '95*, pp. 86–95, 1995.

[6] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.

[7] B. S. Baker. Finding clones with Dup: Analysis of an experiment. *IEEE Trans. Softw. Eng.*, 33(9):608–621, 2007.

[8] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. of METRICS '99*, pp. 292–303, 1999.

[9] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *Proc. of WCRE '99*, pp. 326–336, 1999.

[10] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of WCRE 2000*, pp. 98–107, 2000.

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

[12] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *Proc. of ESEC/FSE 2005*, pp. 156–165, 2005.

[13] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of ICSM '98*, pp. 368–377, 1998.

[14] K. H. Bennet. Software maintenance: A tutorial. In M. Dorfman and R. H. Thayer eds., *Software Engineering*. IEEE Computer Society Press, 1997.

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

[16] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Sofw. Eng.*, 20(6):476–493, 1994.

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

[18] S. Counsell, R. M. Hierons, R. Najjar, G. Loizou, and Y. Hassoun. The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph. In *Proc. of TAIC-PART 2006*, pp. 181–192, 2006.

[19] I. Dagan, L. Lee, and F. C. N. Pereira. Similarity-based models of word cooccurrence probabilities. *Machine Learning*, 34(1-3):43–69, 1999.

[20] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of ICSM '99*, pp. 109–118, 1999.

[21] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[22] G. Fischer and J. W. v. Gudenberg. Simplifying source code analysis by an xml representation. *Softwaretechnik Trends*, 23(2):49–50, 2003.

[23] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[24] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proc. of ICSE 2008*, pp. 321–330, 2008.

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

[26] GNU grep. `http://www.gnu.org/software/grep/`.

[27] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proc. of IEEE ICDM Workshop on Frequent Itemset*, 2003.

[28] D. Gusfield. *Algorithms on Strings, Trees, And Sequences.* Cambridge University Press, 1997.

[29] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):435–461, 2008.

[30] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous modification support based on code clone analysis. In *Proc. of APSEC 2007*, pp. 262–269, 2007.

[31] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notice*, 39(12):92–106, 2004.

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

[33] IEEE Std 1219: Standard for software maintenance, 1998.

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

[35] ISO/IEC 14764:2006 - software engineering – software life cycle processes – maintenance, 2006.

[36] JBoss. http://www.jboss.org.

[37] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE 2007*, pp. 96–105, 2007.

[38] JUnit. http://www.junit.org.

[39] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[40] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proc. of ICSM 2002*, pp. 576–585, 2002.

[41] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proc. of ISESE 2004*, pp. 83–92, 2004.

[42] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. of SAS 2001*, pp. 40–56, 2001.

[43] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. of WCRE 2001*, pp. 301–309, 2001.

[44] S. Kullback. *Information Theory and Statistics*. John Wiley and Sons, 1959.

[45] G. M. L. Prechelt and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[46] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. of ICSM '97*, pp. 314–321, 1997.

[47] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[48] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of ESEC/FSE 2005*, pp. 306–315, 2005.

[49] J. Lin. Divergence measures based on the shannon entropy. *IEEE Trans. Inf. Theory*, 37(1):145–151, 1991.

[50] J. I. Maletic, M. L. Collard, and A. Marcus. Source code files as structured documents. In *Proc. of IWPC 2002*, pp. 289–292, 2002.

[51] J. Mayland, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of ICSM '96*, pp. 244–253, 1996.

[52] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.

[53] T. M. Meyers and D. Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Trans. Softw. Eng. Methodol.*, 17(1):1–27, 2007.

[54] W. F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

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

[56] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *Proc. of OOPSLA 2004*, pp. 432–448, 2004.

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

[58] D. Seipel, M. Hopfner, and B. Heumesser. Analyzing and visualizing PROLOG programs based on XML representations. In *Proc. of WLPE 2003*, pp. 31–45, 2003.

[59] N. Shi and R. A. Olsson. Reverse engineering of design patterns from Java source code. In *Proc. of ASE 2006*, pp. 123–134, 2006.

[60] K. Tanaka-Ishii and H. Iwasaki. Clustering co-occurrence graph based on transitivity. In *Proc. of WVLC '97*, pp. 91–100, 1997.

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

[62] V. Wahler, D. Seipel, J. W. v. Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proc. of SCAM 2004*, pp. 128–135, 2004.

[63] M. Weiser. Program slicing. In *Proc. of ICSE '81*, pp. 439–449, 1981.

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

[65] S. W. L. Yip and T. Lam. A software maintenance survey. In *Proc. of APSEC '94*, pp. 70–79, 1994.

[66] A. Zeller. *Why Programs Fail.* Morgan Kaufmann Pub., 2005.

[67] 井上, 神谷, 楠本. コードクローン検出法. コンピュータソフトウェア, 18(5):47–54, 2001.

[68] 上田. クラスター分析. 朝倉書店, 2003.

[69] 北, 津田, 獅々堀. 情報検索アルゴリズム. 共立出版, 2002.

[70] 齋藤, 宿久. 関連性データの解析法–多次元尺度構成法とクラスター分析法–. 共立出版, 2006.

[71] 日本語入力システム “かんな”. http://canna.sourceforge.jp.

[72] 廣田, 佐々木. 用語解説 “f 値”. 日本ファジィ学会誌, 12(3):36, 2000.

[73] 松尾, 石塚. 語の共起の統計情報に基づく文書からのキーワード抽出アルゴリズム. 人工知能学会論文誌, 17(3):217–223, 2002.

[74] 横森, 梅森, 西, 山本, 松下, 楠本, 井上. Java ソフトウェア部品検索システム SPARS-J. 電子情報通信学会論文誌 D-I, J87-D-I(12):1060–1068, 2004.