ソフトウェアの類似性の分析とその応用に関する研究

川口 真司

2006 年 1 月

内容梗概

ネットワーク環境の発展に伴ない開発者が地理的にさまざまな場所に分かれたままソフトウェア開発を行う分散開発環境が実際の開発においても採用されるケースが増加している．この分散開発環境を支えるシステムとしてさまざまなシステムが提案，実装されているが，その中でもソフトウェアリポジトリは最も重要な役割を果している．ソフトウェアリポジトリはネットワーク上にてソフトウェア開発の進展にしたがって生成されるさまざまな成果物を一括管理，保存するためのシステムである．近年，ソフトウェアリポジトリは活発に利用されるようになっており，それだけ大量のデータがソフトウェアリポジトリに蓄積されるようになっている．例えばオープンソース開発において利用されている最も著名なソフトウェアリポジトリである SourceForge は 100 万を越えるプロジェクトに利用されている．

また計算機資源の発展に伴い，膨大なデータから統計に基づいて特徴的なデータや類似データを発見するデータマイニングと呼ばれる手法が注目を集めている．これらの手法は大規模なデータの存在を前提としているため従来ソフトウェア工学の分野においては適用が難しかったが，ソフトウェアリポジトリの台頭によりデータマイニング手法の適用に必要な大規模なデータの入手が現実的なものとなった．

そこで本論文ではソフトウェアのもつ類似性に着目して，ソフトウェア開発工程の改善に有用な情報の抽出を行う手法の提案，実装を行う．具体的には特に以下の問題点に着目する．

1. 既存手法が着目する類似度取得対象が細粒度すぎること

2. 既存の細粒度な類似度取得手法は過去の類似度が考慮されていないこと

まず 1. で触れている問題点であるが，従来の手法では主にソフトウェアの構成要素であるソフトウェアコンポーネントや，コードクローンに着目しており，ソフトウェアそのものについてはあまり取りあげられていない．ソフトウェアそのものに関する類似度は，ソフトウェアリポジトリに集積された大量のソフトウェアを分類するために必要不可欠といえる．そこでソースコードを利用してソフトウェア間の類似度を定義し，ソフトウェアの分類を行う手法を提案する．本手法はソースコードに対して潜在的意味解析手法 LSA を適用することで分類を行う．LSA は

元々自然言語で記述された文書を分類するための手法であり，単語の出現頻度を統計的に解析することによって精度の高い分類を実現している．本手法の特長はソースコードのみを利用し，付随ドキュメントに依存しないこと，分類先であるカテゴリも自動的に抽出するため事前知識が必要ないこと，そして一つのソフトウェアが複数のカテゴリに分類されることを許す非排他的な分類を行う点である．また，提案手法によって分類した結果を効率的に閲覧するためのシステムを実装する．提案手法では一つのソフトウェアは複数のカテゴリに分類されうるため，従来のツリー型のような描画方法では閲覧が困難である．そのために Unifiable Cluster Map 手法を提案，実装を行った．本手法は Cluster Map 手法を大規模なデータに対応するよう拡張したものである．そして提案手法の分類結果は，適用の際に必要な情報量が少ないにもかかわらず同程度の分類精度を実現していることを確認した．

次に 2. で触れている問題点について述べる．ソフトウェア中に出現する重複コードであるコードクローンを抽出するために，さまざまな手法が提案されている．ほとんどのコードクローン手法では，一字一句一致する重複コードだけではなく，ある程度変更が加わっている重複コードもまたコードクローンとして抽出する．そうすることでコピーされたあとに少々の変更が加えられたとしてもコードクローンとして検出できるようにしている．しかし，大きな変更も許容してしまうと無関係な部分までコードクローンと誤認識してしまうため，あまり許容幅を広げられない．したがって，過去に重複コードであった部分も変更が加わるにつれて検出が困難になる．そこで，コードクローン履歴という形でコードクローンの過去を分析することによって，このような関連を抽出する．そして実際に抽出された過去にコードクローンであったコード片には強い関連があることを実験によって確認した．また，過去のコードクローンを分析することによって得られるクローン量の変化をグラフ化することによってコードクローンが急激に増加していないかどうかを監視することが可能になる．

論文一覧

主要論文

[1-1] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, Katsuro Inoue: ”Automatic Categorization Algorithm for Evolvable Software Archive”, Sixth International Workshop on Principles of Software Evolution (IWPSE2003), pp-195-200, Helsinki, Finland, September 1-2, 2003. (国際会議)

[1-2] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, Katsuro Inoue: ”MUDABlue: An Automatic Categorization System for Open Source Repositories”, The 11th Asia-Pacific Software Engineering Conference(APSEC2004), Busan, Korea, November 30th - December 3rd, 2004. (国際会議)

[1-3] 川口真司, パンカジ ガーグ, 松下誠, 井上克郎: ”MUDABlue: ソフトウェアリポジトリ自動分類システム”, 電子情報通信学会論文誌 D-I, Vol.J88-D-I, No.8, pp.1217-1225, 2005.(学術論文)

[1-4] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, Katsuro Inoue: ”MUDABlue: An Automatic Categorization System for Open Source Repositories”, The Special Issue of Journal of Systems and Software.[採録決定] (学術論文)

[1-5] 川口真司, 松下誠, 井上克郎: ”版管理システムを用いたコードクローン履歴分析手法の提案”, 電子情報通信学会論文誌 D-I [投稿中] (学術論文)

関連論文

[2-1] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, Katsuro Inoue: ”Automatic Categorization Tool for Open Software Repositories”, The 1st Workshop on Open Source in an Industrial Context (OSIC ’03), pp.11-16, Anaheim, California USA, October 26, 2003. (国際会議)

[2-2] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, Katsuro Inoue: ”On Automatic Categorization of Open Source Software”, 3rd Workshop on Open Source Software Engineering: Taking Stock of the Bazaar(ICSE2003), Portland, Oregon, USA, May 3, 2003. (国際会議)

謝辞

本研究の全般に関し，常日頃より適切なご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，適切なご助言とご指導を頂きました，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，同 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，適切なご助言とご指導を頂きました，大阪大学大学院情報科学研究科萩原 兼一 教授，八木 康史 教授に感謝致します.

本論文を執筆するにあたり，直接具体的なご指導を頂きました，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 松下 誠 助教授に心より感謝致します.

本研究を行うにあたり，ご助言やご指導を頂きました，産業技術総合研究所 神谷 年洋 氏，Zee Source 社 Pankaj K. Garg 氏に感謝致します.

最後に，井上研究室の皆様のご助言, ご協力に御礼申し上げます.

目 次

図 目 次

表 目 次

第1章　まえがき

近年，ソフトウェアは社会的基盤の一部にも浸透しており，その重要性は日々増大している．またソフトウェアの担う責務の膨大さから，ソフトウェアの大規模化，複雑化は避けられず，ソフトウェア開発に要する費用は増加の一途を辿っている．このように高品質なソフトウェアを低コストで開発するという相反する要望を満たすことを目的としたソフトウェア工学に関する研究が行われてきた．

特に近年は大量のソフトウェアを保持・管理するソフトウェアリポジトリが活発に利用されており，そこに蓄積された大量のデータを用いることによって，大規模なサンプルから統計的に類似性を判定する手法の適用可能性が広がっている．これらの手法はデータマイニングと呼ばれ計算機資源の飛躍的発展にともない種々の手法が実用化されている．

ソフトウェア工学において行われた研究にはさまざまなアプローチが存在するが，本研究ではソフトウェアの持つ類似性に着目した手法に焦点を当てる．類似性に着目した手法とは，例えばソフトウェアの中から類似したソフトウェア部品を抽出するなど，ソフトウェアを開発する際に生成される各種プロダクトから類似したプロダクトを発見することで開発作業の分析や支援を行うアプローチである．

ソフトウェアは実際には様々な階層から構成されているため，類似度を測定する手法もそれぞれの階層を対象にしたものが提案されている．まず最も細い単位としてはソースコードに含まれる単語がある．単語がいくつかつらなったものはコード片とよばれる．次に大きな単位としてはブロック，手続き，関数などのプログラミング言語によって規定される構成要素が挙げられる．さらにそれらの集合であるクラス，モジュール等の単位がある．これらの類似性は，類似した部品を発見することで再利用のための部品探索や，重複箇所の削除などの作業を支援などに利用される．

以下，それぞれの階層についての類似性に着目し，ソフトウェア開発の支援を行う研究について取りあげる．

1.1 類似性に着目した既存手法

1.1.1 コードクローン

コードクローンとは，ソフトウェアの中で繰り返し出現するソースコード片のことを指す．コードクローンがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものがある [11, 33].

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，ゼロからコードを書くよりよりも既存コードをコピーして部分的な変更を加える方が信頼性が高いということもあり，実際には，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存在する．

コーディングスタイル 規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

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ければ，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場合がある．

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．

偶然 単純に偶然一致してしまう場合もあるが，大きなコードクローンになる可能性は低い．

ソフトウェアの内部において重複したコードを抱えることは工数の無駄であるばかりか，後の保守工程において工数を上げる要因となることが指摘されている．もしコードクローンの一つに欠陥がみつかった場合，全てのコードクローンを検

出し，同じ修正を加えなければならないからである．このとき，もしコードクローンがソフトウェアのさまざまなところに散らばっていたら，それら全てを漏れなく検出することは大変な労力を要する．ソフトウェアが小規模ならば開発者の手で漏れなく検出することも可能であるが，数百万行，数千万行の規模の大規模なソフトウェアに対して漏れなく検出することは事実上不可能である．

この問題に対処するべく，これまでにクローンを検出するための様々な手法が提案されており，そのいくつかは実際に利用可能なシステムとして実用化されている [14]．このようなクローン抽出システムは，大規模なソフトウェアから自動的にクローンを発見し，ユーザに提示する．開発者は発見されたクローンを適切な方法で抽象化することでソフトウェアの品質を高めることができる．

クローン検出手法には大きくわけてソースコードの字句解析に基づく手法 [6, 7, 5, 11, 20, 33, 38, 39, 42, 52, 54] と，特徴メトリクスに基づく手法 [8, 9, 10, 32, 48, 50] に分けられる．ソースコードの字句解析に基づく手法 では，ソースコード中で同一の文字列を検索することでクローンの検出を行う．ただしコード片がコピーされるときには，若干の変更が加えられることが多い．そのため，まったく同一の文字列のみを検索するのではなく，若干のゆらぎを許容する形で検索を行う．特徴メトリクスに基づく手法では，例えばクラスや関数，ファイルのようなプログラム中のある種の単位ごとに特徴メトリクスを定義・算出し，それらのメトリクス値が類似したものをクローンとして抽出する手法である．これらの手法は字句解析に基づく手法と比べて高速であり大規模なソフトウェアへの適用も容易である反面，比較対象が固定されているため，より細かい単位で存在するクローンを発見することは難しい．それぞれの手法，ツールの特徴は次のようになっている．

Covet

文献 [48] 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 [11]

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，Progress である．

Dup [6, 7, 5]

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 [52]

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 [38]

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検出方法では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．[38] 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 [39]

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 [8, 9, 10]

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 [54]

JPlag 同様，プログラム盗用の検出を目的として開発された．ソースコードか

ら空白を取り除いた後に，ソースコードの n-gram からハッシュ値を計算し，その値を利用して同一文字列の発見を行う．検出対象言語は，Ada，C/C++，Java，Lisp，ML，Pascal，Scheme 等，26 種類に及ぶ．

CP-Minder [42] ソースコードを字句解析し，文単位でハッシュ値の計算を行う．そして，ブロック単位でクローンの検出を行う．検索対象言語は C/C++ である．

Merlo らの手法 [50] ソフトウェア中に各クラスに対してメトリクス値を計算し，それらが一致したものをクローンと検出する．検索対象言語は Java である．

いずれの手法，ツールにおいても提案者によってコードクローンの定義が微妙に異なっており，検出されるコードクローンが異なっている．つまり，コードクローンの定義とは検出アルゴリズムそのものによって定義される．Burd ら [14] も，CloneDR，Covet，JPlag，Moss，CCFinder の以上 5 つのツールを用いて，それぞれ検出されるコードクローンの比較が行っているが，全ての面において他のツールよりも優れているツールはなく，使う場面に応じて，適切なツールを選ぶことが必要となってくると述べている．

これらの提案手法を利用することによって，開発者は削除するべきコードクローンを自動的に発見することができる．こうして発見されたコードクローンは基本的に削除するべきであるが，プログラムの構成によっては削除が困難であったり，非常に煩雑なものも少なくない．そこで，発見されたクローンの削除を支援する手法も研究されている．その他にも重複コードを再利用される可能性が高い部品とみなしてこれをソフトウェア部品として抽出するような試みなどがある．

1.1.2 ソフトウェアコンポーネントの再利用

クローンが対象としているコード片よりも粒度の粗い単位として，類似性に関する研究が活発に行われているものとして，ソフトウェアコンポーネントが挙げられる．ソフトウェアコンポーネントは主に再利用の対象として着目されることが多い．ソフトウェアコンポーネントの再利用とは，全てを一から開発するのではなく，既存のソフトウェアコンポーネントを組みあわせて開発を行う試みである．ソフトウェアコンポーネントの再利用がもたらす利点を以下に挙げる．

- 低コスト

  すでにテストが終了しているソフトウェアを利用することで，そのソフトウェアの設計，実装，テストに必要な工数を省略することが可能となる．これは，一からすべてを開発する場合と比較して大幅なコスト削減となる．

- 高信頼性

  再利用するソフトウェアはすでにテストが終っており，また実際に利用に耐える品質であることがすでに示されている．そのため再利用するソフトウェアが担当する部分については非常に高い品質となる．また，そのソフトウェアを再利用されるケースが増えれば増えるほど，それだけ欠陥修正の機会は増え，ますます品質が高くなることが期待できる．

- 標準化

  ソフトウェアを再利用しやすいように構成することは，ソフトウェアをある一定の型式に従って作成することを要請する．このことは開発されるソフトウェアの標準化を促す．

ソフトウェアの再利用は実際にはいくつかの工程に分割される．まず最初に必要な工程は再利用の対象となる部品を収集する作業である．近年では系統的にソフトウェアコンポーネントを蓄積する組織も増えつつあるが，特に初期段階においてはコンポーネントが存在しない，もしくは十分な数を揃えられない場合も多い．そこでこれまでに開発したソフトウェアのなかから再利用可能なコンポーネントを抽出する手法が提案されている．これらの手法では前節で述べたクローン抽出手法や，独自に定義したコンポーネント間の類似度に基づいてコンポーネントを発見する．

さまざまな手法で抽出されたコンポーネントは通常，一箇所に集められ利用者に提供される．ソフトウェアコンポーネントが数百，数千個という単位で集積されると，何らかの手段でコンポーネントを分類・抽出する手法が必要になる．ここでもコンポーネントを自動的に分類するためにコンポーネント間の類似度が使われる．

最後に再利用コンポーネントを実際に適用させる．見つかったコンポーネントがそのまま利用できればよいが，希望する動作と少し異なることも多い．そのような場合にはコンポーネントを修正することが必要となる．

またコンポーネント間の類似度に着目した研究には，ソフトウェアクラスタリングと呼ばれる研究が存在する．これはソフトウェアの理解を容易にするために，ソフトウェアをいくつかのサブシステムに分割しようという試みである．そのために，ソフトウェアを構成するコンポーネント間の類似度を用いて，各コンポーネントをいくつかのグループに分類している．

1.2 ソフトウェアリポジトリ

これまでに，クローンや類似ソフトウェアコンポーネント解析手法，およびそれらの応用手法について議論したが，そのためには検索対象が何らかの形で保存されていることが保証されていなければならない．

近年，ネットワークの発達および分散ソフトウェアの一般化に伴ないネットワーク上にソフトウェアプロダクトを保存するソフトウェアリポジトリが急速に普及している．特にオープンソースの世界では，SourceForge [55] を始め，さまざまなソフトウェアリポジトリがある．2004 年 12 月現在，SourceForge を利用しているプロジェクトは 9 万を越えており，現在も急速に増加しつづけている．さらに，企業内プロジェクトを共有するために，社内向けサービスとしての導入も活発に行われている．さらに，いくつかの企業では社内プロジェクトを管理するためにソフトウェアアーカーブサービスを設置している．Hewlett-Packard Company や IBM, Motorola，Nokia，Xerox などはそのような企業の一例である．

ソフトウェアリポジトリを導入するメリットには以下のような点が挙げられる．

- ソフトウェア開発における基盤環境を容易に調達することができる．プロジェクト名など最低限の情報を入力するだけで新規プロジェクトの立ちあげが可能になる．
- ソフトウェアリポジトリに格納された膨大なソフトウェアから，希望するソフトウェアを検索し，利用することができる．
- 開発者が開発中のソフトウェアと類似したソフトウェアを検索することで，検索したソフトウェアを再利用したり，開発者間での情報共有を図ることが可能になる．ソフトウェアリポジトリには最終成果物であるプログラムそのものだけでなく，ソースコードやメーリングリストのログ，バグ情報データベース等，開発中にやりとりされた情報も記録されている．このような情報もまた，開発者にとっては非常に有用である．
- 管理者が管理下にあるプロジェクト全体を把握する．社内にソフトウェアリポジトリを導入した場合，社内のすべてのプロジェクトをソフトウェアリポジトリにおいて管理することが可能である．そのため，管理者はリポジトリ全体を俯瞰し，現状を把握することが可能となる．

ソフトウェアリポジトリはソースコードを管理する版管理システムをはじめ，各種ドキュメントやメーリングリストのログ，開発用の掲示板やバグレポート，バイナリパッケージ配布用ページ等が用意されている (図 1.1).

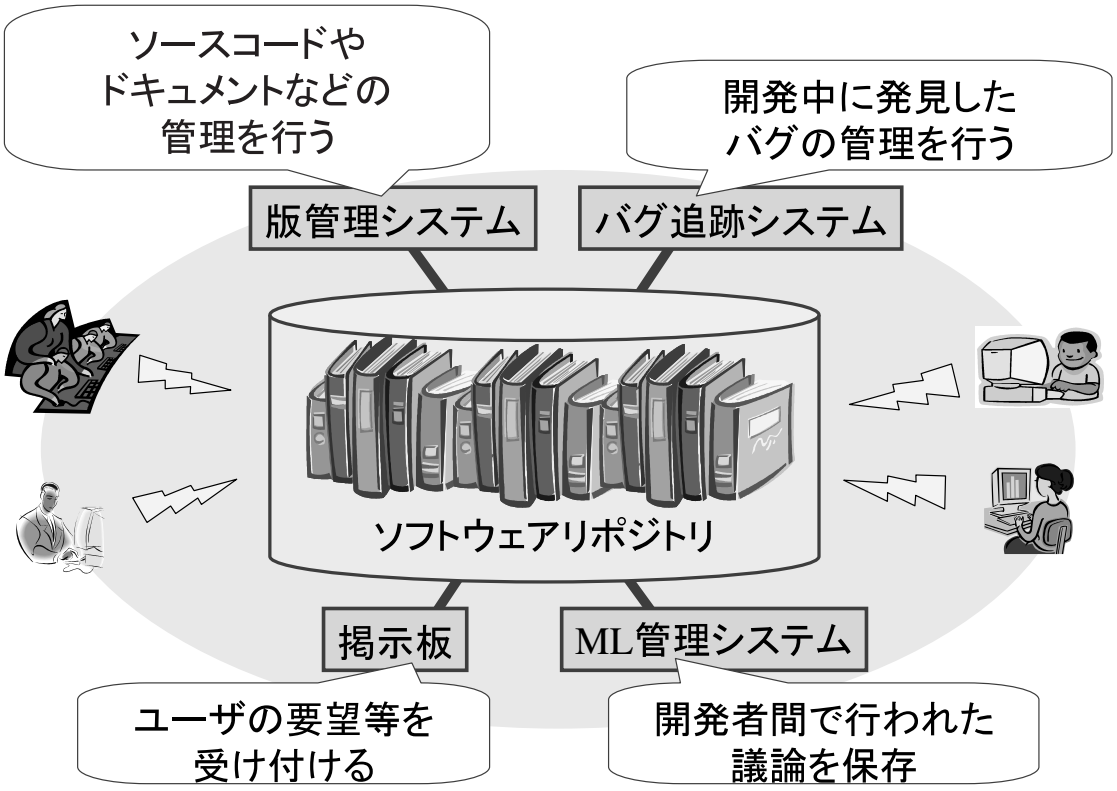


図 1.1: ソフトウェアリポジトリ

その中でもプロダクトの管理を行う版管理システムはソフトウェアリポジトリの中でも中心的役割を担うものである．版管理システムは一般に以下にあげる機能を実装している．

- 一元管理

  ソフトウェアリポジトリによってプロダクトを一元管理することで，複数の正本が作られてしまうという事態を避けることができる．

- 多人数でのスムーズな共有が可能

  一元管理を容易にするために，共同開発者の更新内容を自動的に反映させる機能など，さまざまな機能が提供されている．そのため開発者はプロダクトの管理に煩わされることなく開発に専念することができる．

- ネットワーク越しに利用可能

  現代の版管理システムの多くはネットワークを通じてのプロダクト取得，更新をサポートしている．そのため開発者は地理的な制約を一切受けずに開発を進めることが可能である．

- 過去へのロールバックの保証

  ソフトウェアリポジトリでは一つ一つの更新を全て記録しているため，逆にそれらの更新を取りけして，過去の時点に戻すことができる．ソフトウェア開発においては様々な試行錯誤がつきまとうため，任意の時点の状態にプロダクトを差し戻す機能は必要不可欠とも言っていいほど重要な機能である．

1.3 既存の手法の問題点

これまでに類似性の分析を用いたさまざまな研究について述べた．本研究では以下の 2 点に着目する．

- ソフトウェアそのものの類似度が考慮されていない

  コンポーネントを単位とする再利用は主にソフトウェアの詳細設計や実装を行う際の支援を主な目的としている．しかし，ソフトウェアコンポーネントをどのように組みあわせるのか，ソフトウェア全体をどのように構成するのか，といった知見はコンポーネントから得ることは難しい．概要設計のレベルについて理解を深めたい場合には，コンポーネントより大きな単位であるソフトウェアそのものの再利用も考慮する必要がある．

  そのためには，先行して作成されたソフトウェアの集合から，これから作成しようとしているソフトウェアに類似したものを検索するための仕組みが必要となる．類似ソフトウェア検索を有効に活用することで，車輪の再発明を防ぐことができる．また，たとえ発見できたソフトウェアそのものを直接再利用するのが困難だったとしても，開発中に議論された内容のログを見たり，そのソフトウェアの開発者にインタビューしたりすることによって知見を得ることができる．

- クローンの過去が考慮に入っていない

  各種クローン検出技法では，クローンとして検出されるコード片にある程度の幅をもたせている．そうすることで多少の編集が加わっていてもクローンとして検出するようにしている．ただし，あまりに基準を緩くするとクローンとはいい難いものまでクローンと判定してしまい，逆に基準を厳しくするとクローンと判定するべきコード片を漏らしてしまう．

  このようにコード片の類似基準を変更して適正なクローンを判定するのは限界がある．そこで，コード片の過去に着目することを考える．すなわち，現時点のソースコードだけではなく，過去に遡ってクローン検出技法を適用す

ることで，過去のどこかの時点でクローン関係にあったコード片を検出することができる．このような関係も考慮に入れることで，より柔軟なクローン検出が可能となる．

1.4 本論文の概要

本研究ではリポジトリに含まれる大量のデータを整理・統合してソフトウェア開発支援を行うことを目標とする手法を提案し，その評価を行う．本論文では，これまでに提案した二つの手法について述べる．

1. MUDABlue: ソフトウェアリポジトリ自動分類システムの構築

   ソフトウェアリポジトリは膨大な数のプロジェクトを保持しているため，例えば，開発者が現在開発中のものと似ているプロジェクトを捜したり，管理者が会社内で走っている全プロジェクトを俯瞰するといったことに活用できる．しかし，保持内容が膨大なためにプロジェクト同士の関連を判定して整理するには非常な労力を必要とする．そこで，我々はソフトウェアを自動的に分類する MUDABlue システムを作成した．MUDABlue の特長は以下の 4 つである．1) 分類にはソースコードのみを使用，2) 分類先となるカテゴリ集合も自動的に決定する，3) ソフトウェアを二つ以上のカテゴリに分類することを許す，4)Web インタフェースで分類結果を表示する．本論文では既存の分類手法との比較を通じて MUDABlue システムの有効性を議論する．

2. クローン履歴関係抽出手法の提案

   既存のクローン検出手法は，ある時点におけるソースコードを対象にクローンを検出する．そのため，クローンがどのような過程で生まれ，どのような変遷を辿ったのかは一切考慮されない．しかし，クローンの履歴を用いることで，かつてクローン関係にあったコード片を抽出してクローン検出の漏れをなくすことができる．また連続的にクローン分析を行うことで，クローンの行数や，その割合の変化を時系列にそったグラフとして表現できる．このグラフは過去の開発においてクローンが増えた時点に着目したり，急激なクローン量の変化を視覚的にとらえることを可能にする．そこで本論文では，版管理システムに蓄積されたソースコードを対象としてクローンの履歴を抽出する手法を提案する．このクローン分析を過去の時点に遡って順次適用することでクローンの履歴を抽出する．また，PostgreSQL に対して提案手法を適用し，抽出できるクローンの有用性や，実際に得られたクローン量グラフについて考察を行う．

以下，第 2 章ではソフトウェア自動分類システム MUDABlue について述べる．第 3 章ではクローン履歴関係抽出手法について述べる．最後に第 4 章で本論文の研究についてまとめ，今後の研究方針を述べる．

第2章　MUDABlue: ソフトウェアシステム自動分類手法

2.1　導入

近年，オープンソース開発においては SourceForge を始めさまざまなソフトウェアリポジトリが活用されている．また企業においても，社内プロジェクトを管理するためにソフトウェアリポジトリの導入が進んでいる．

一般にソフトウェアリポジトリには膨大なプロジェクトが登録されているため，実際に希望するソフトウェアや類似ソフトウェアを検索するためには，ソフトウェアが分類，整理されていなければならない．現存するリポジトリサービス (例えば SourceForge，Freshmeat [24]，Tigris [59]，SourceShare [56]，GForge [25] など) では，プロジェクトを登録する人がそのプロジェクトの分類を行う．しかし，このような手動分類では 2 つの問題点が存在する．

カテゴリ定義の労力 カテゴリ分類を実現するには，まず分類先となるカテゴリ集合を定義しなければならない．しかし，よいカテゴリ集合を定義するには，格納されているソフトウェアやそれらの用途，依存しているライブラリ等について精通していなければならない．このような分類を一人で作りあげるのは，リポジトリの規模を考えると相当困難である．また複数人で作成する場合にはカテゴリ分けについて矛盾が生じる危険がある．

非一貫性 個々のソフトウェアがどのカテゴリに分類されるのかは，分類を行う登録者に依存する．そのため，カテゴリ作成者の意図とは異なるカテゴリが選択される可能性がある．そのため，本来同一のカテゴリであるべきソフトウェア同士が異なる分類になってしまったり，逆に異なるカテゴリであるべきソフトウェアが同一の分類になってしまう．

このような問題を解消するために，我々は自動的にソフトウェアの分類を行う MUDABlue システムの提案を行う．MUDABlue は IR 手法[1]の一種である 潜在的意味解析手法 (Latent Semantic Analysis．以下 LSA) を用いて，カテゴリ集合の作

[1] IR: Information Retrieval

成とソフトウェアの分類を自動的に行う．MUDABlue はソースコードのみに依存するため，管理者の入力は一切必要としない．LSA は単語間の意味的繋がりを統計的手法を用いることで抽出する手法である [41].

MUDABlue は分類した結果を Web インタフェースを通じてユーザに提供する．2.2 で述べるとおり MUDABlue は一つのソフトウェアに対して複数カテゴリを割りあてることを許す非排他的な分類を行う．分類結果を描画するために，我々は Cluster Map 手法 [22] を元にした Univiable Cluster Map 手法を定義する．本手法を用いることでソフトウェアリポジトリ全体を容易に閲覧することが可能となる．

以下，2.2 にてソフトウェア分類手法が備えるべき特性について考察し 2.3 にて提案する MUDABlue 分類手法について述べる．そして 2.5 にて MUDABlue 分類手法について実験を行い， 2.6 でその実験結果について考察を述べる．最後に関連文献について 2.7 で触れたのちに 2.8 で本論文のまとめを行う．

2.2 ソフトウェア分類

ソフトウェア自動分類手法は一般に

1. 分類先となるカテゴリ集合を定義する工程
2. 各ソフトウェアをカテゴリに割りあてる工程

以上 2 つの工程からなる．

既存の自動的分類に関する研究 [45, 61] では (2) の工程の自動化のみに着目しており，カテゴリ集合についてはリポジトリ管理者から与えられることが前提となっている．我々は (1) のカテゴリ集合を作成する工程の自動化も重要であると考える．カテゴリ集合の定義は，カテゴリ結果に重大な影響があること，そしてカテゴリ集合の作成もまた労力のかかる作業だからである．

これまでの手動分類においては，たいていソフトウェアは用途によって分類される．しかし，我々はソフトウェアを依存しているアーキテクチャや利用しているライブラリによって分類することも，また有効であると考える．例えば，ソフトウェアのアーキテクチャ(GUI アプリケーション，CUI アプリケーション，サーバアプリケーションなど)や，利用するライブラリ (Microsoft Foundation Class Library(MFC), Gnome Tool Kit(GTK)，正規表現ライブラリなど) が考えられる．このようなカテゴリも，あわせて使うことで検索作業がより便利になる．

また既存の研究 [45, 61] はソフトウェアを単一のカテゴリに分類する．そのため，前述のような様々な側面を考慮した分類を行うことが困難である．図 2.1 では，software1 と software2 はエディタである．さらに，software1 と software3 は

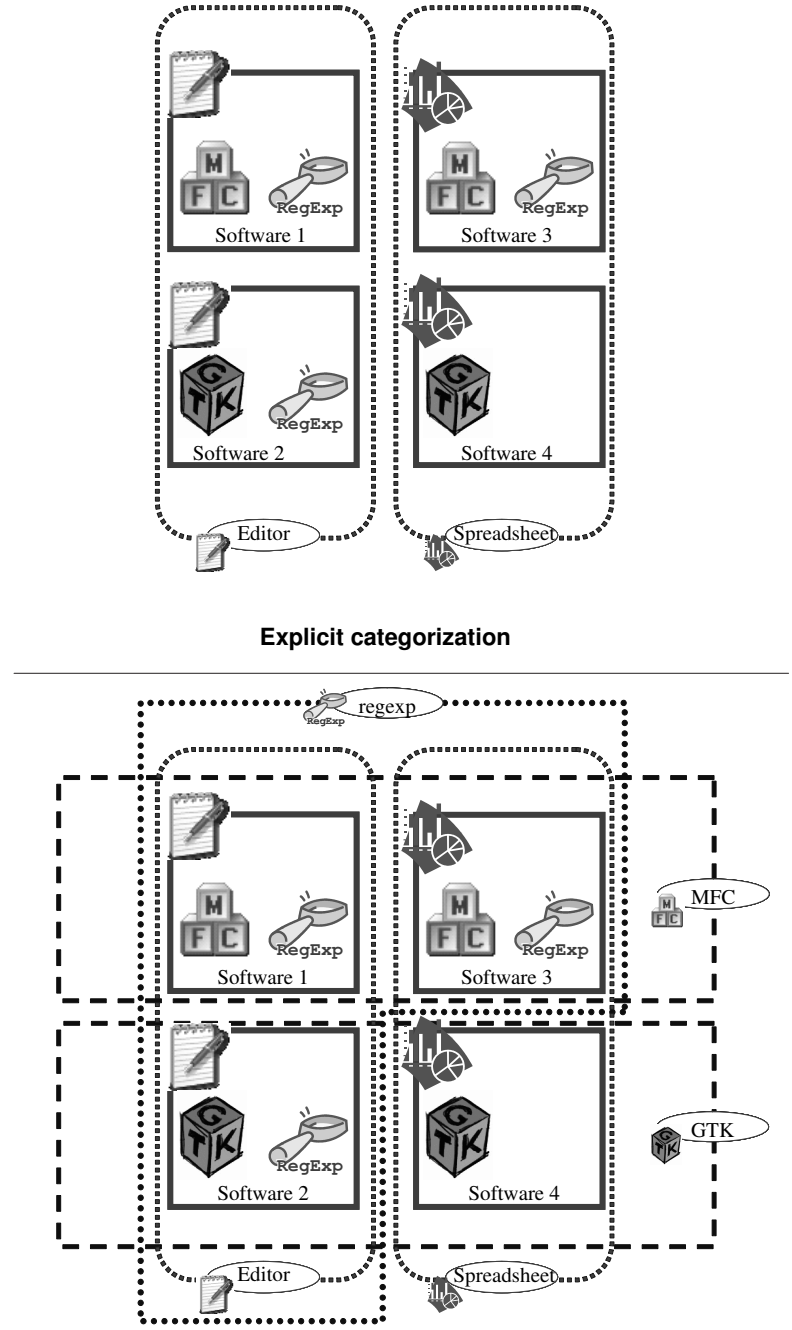


図 2.1: 非排他的分類

MFC ライブラリを用いて実装されている．このような場合，software1 は Editor カテゴリと MFC カテゴリの両方に所属する形になる．一つのソフトウェアについて，単一のカテゴリのみへの分類を行う手法ではこのような分類を行うことはで

きない．我々は，これまでにソフトウェア間の類似度を測定することでソフトウェアの分類を試みている [34]．この研究では，我々は LSA を用いてソフトウェア間の類似度の測定を行った．その結果得られた類似度は，もっとも特徴的な部分のみしか反映されていないことがわかった．例えば，GTK で実装されたユーザインタフェースを含むデータベースと GTK で実装されたエディタは，その使用用途が全く異なるにも関わらず，高い類似度を示した．この結果からも，分類を行う際にはソフトウェアのさまざまな側面を考慮に入れる必要があることがわかる．

さらに，既存の自動分類に関する研究ではソフトウェアに付随する各種文書に対して解析を行い，分類を実現している．しかし開発文書はソフトウェアによって質，量が大きく異なる．特に開発初期のプロジェクトでは文書が全く存在しないことも多い．そのため，どのようなプロジェクトにも存在するソースコードを用いたほうが，より適用範囲が広くなる．

これらのことを踏まえて，自動ソフトウェア分類に求められる特性として以下の 3 点をあげることができる．

1. カテゴリ集合も自動的に抽出する
2. ソフトウェアが複数のカテゴリに分類されるのを許す
3. ソースコードのみに依存する

2.3 MUDABlue

MUDABlue システムは大きくわけてカテゴリ抽出，分類を行う分類部と，分類結果を利用してカテゴリ，ソフトの検索を行うカテゴリ描画部とで構成される．本節では，それぞれのサブシステムについて概要を述べる．

2.3.1 分類手法

MUDABlue は 2.2 で述べた 3 つの特性を備えたソフトウェア自動分類システムである．本節では，まず MUDABlue で用いている IR 手法である LSA について簡単に述べ，その後 LSA を用いたカテゴリ抽出方法の概要，および詳細ステップを述べる．

潜在的意味解析手法 - LSA

LSA は文書群のなかから，単語間の潜在的な関連を考慮して，単語間もしくは文書間の類似度を算出する手法である [41]．LSA では文書内に出現する単語の出

現頻度を表す word-by-document 行列を作成する．そして，この行列に対して特異値分解と呼ばれる数学的操作を行うことで単語間の潜在的関連を抽出する．この操作は word-by-document 行列のみを入力として行われるため，例えば，どの語とどの語が類義語であるとか，文書 A と文書 B は同一カテゴリの文書であるといった学習用データは一切使わない． このように，学習用データを必要としないのは LSA の大きな特徴の一つである．

LSA はデータマイニングの他，人間の知識獲得過程の理解等，さまざまな分野で応用されている [40, 19]．ソフトウェア工学においても，単一のソフトウェアを複数のコンポーネントに分割したり [46]，ドキュメントとソースコード間の結びつきを復元するのに活用されている [47]．以下に LSA の概要を説明する．なお，LSA の詳細については文献 [41] に詳しく述べられている．

ここでは例として，表 2.1 の 6 つの文書があるとする．ベクトル空間モデルでは，これらの文書は表 2.2 のような行列で表される．行列内のセルは文書内での単語の出現頻度を表している．この行列は word-by-document 行列とよばれる．そして，各列を文書ベクトル，各行を単語ベクトルとよぶ．

c1	Human machine interface for ABC computer applications
c2	A survey of user opinion of computer system response time
c3	Relation of user perceived response time to error measurement
m1	The generation of random, binary, ordered trees
m2	Graph minors IV: Widths of trees and well-quasi-ordering
m3	Graph minors: A survey

表 2.1: 文書サンプル

	c1	c2	c3	m1	m2	m3
computer	1	1	0	0	0	0
user	0	1	1	0	0	0
response	0	1	1	0	0	0
time	0	1	1	0	0	0
survey	0	1	0	0	0	1
trees	0	0	0	1	1	0
graph	0	0	0	0	1	1
minors	0	0	0	0	1	1

表 2.2: word-by-document 行列の例

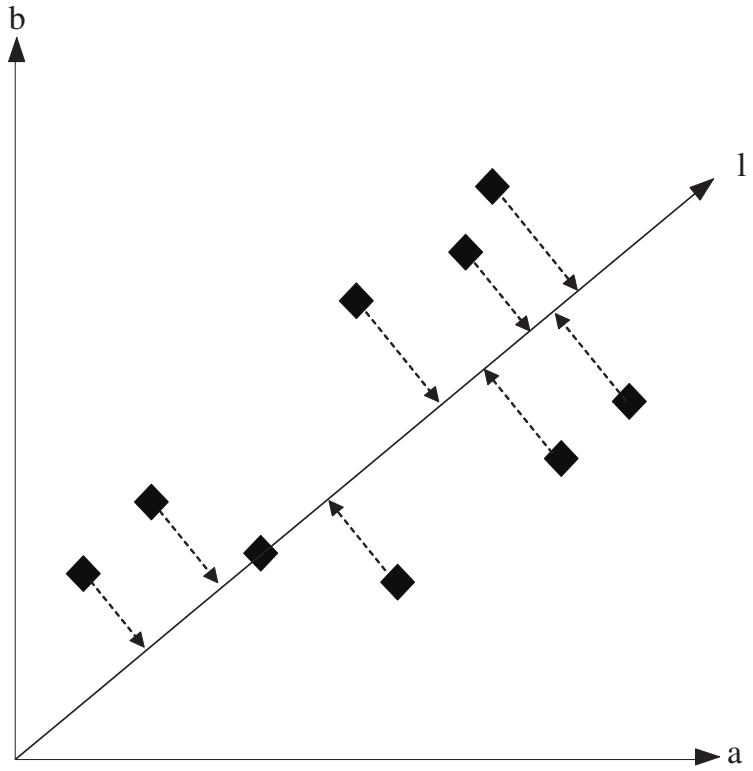


図 2.2: 次元削減

	C1	C2	C3	M1	M2	M3
computer	0.12	0.76	0.53	-0.02	-0.02	0.10
user	0.18	1.11	0.78	-0.04	-0.10	0.09
response	0.18	1.11	0.78	-0.04	-0.10	0.09
time	0.18	1.11	0.78	-0.04	-0.10	0.09
survey	0.11	0.75	0.45	0.10	0.46	0.55
trees	-0.02	-0.02	-0.11	0.16	0.64	0.59
graph	0.00	0.08	-0.09	0.24	0.99	0.93
minors	0.00	0.08	-0.09	0.24	0.99	0.93

表 2.3: LSA を施した word-by-document 行列

ベクトル空間モデルでは，文書間，単語間の類似度は文書ベクトル，単語ベクトルを元に決定する．最も一般的な定義は二つのベクトルの cos を文書間，単語間の類似度とする方法である．

しかし，このベクトル空間モデルの大きな問題点として全く同じ単語以外は類似度の計算に影響を及ぼさない点がある．例えば c2 の文書の response time という

	C1	C2	C3	M1	M2	M3
C1	1	0.45	0.00	0.00	0.00	0.00
C2		1	0.77	0.00	0.00	0.26
C3			1	0.00	0.00	0.00
M1				1	0.58	0.00
M2					1	0.67
M3						1

表 2.4: 文書間の類似度 (LSA 適用前)

単語が performance に置きかわっていた場合，c2，c3 間の類似度は非常に小さいものとなってしまう．このようにベクトル空間モデルでは類似語については全く考慮されていない．そこで LSA では特異値分解とよばれる操作を word-by-document 行列に行うことで，この問題の解決を図っている．特異値分解は因子分析の一種であり，行列の次元を最小二乗誤差で削減する．図 2.2 は特異値分解を用いて 2 次元データを新しい軸 l を導入して 1 次元に縮退した例である．このように，次元削減を行うことで，データ群の特徴を保ったままデータ量を削減することができる．さらに，高次元データに対して次元削減を施した場合，単純にデータ量が減るというだけでなく，より類似度の高い単語から順番に一つの次元としてまとめられることが期待できる．すなわち performance という単語と response time という単語が似た性質を持っていたとする．この場合，二つの単語を別々の次元とするのではなく，一つの次元で表現しても，データが持つ特性に与える影響は少ない．そのため，LSA を用いることで，類似度や共通して現れる頻度の高い単語から優先的に同一次元へ統合されることが期待できる．このようにして，類義語や同一カテゴリ内で頻繁に使われる語句の間の共起関係を反映した形での類似度算出が可能となる．

実際に，表 2.2 に対して LSA を適用して次元削減した結果を表 2.3 に示す．また 表 2.2，表 2.3 において文書間の類似度を計算したものを表 2.4，表 2.5 に示す．LSA を施す前の文書間の類似度では c1 と c2，c3 間の類似度は低い値にとどまっているのに対し，LSA を施した後の行列で類似度を計算すると c1 と c2，c3 の類似度は非常に高い値となっている．これだけでなく，全体としても LSA を施した後の行列のほうが鮮明な結果を持っていることがわかる．

なお，実際に LSA を適用する際には，各単語の出現回数を直接用いるのではなく，単語頻度 (tf:Terminal Frequency) の正規化を行ったり, ある単語が文書空間全体の内でその単語がどれだけ出現したか (df: document frequency) を考慮した変換を行う. df は単語の普遍性を表わしており, df が低いほど, その単語は特徴的であ

	C1	C2	C3	M1	M2	M3
C1	1	1.00	1.00	-0.11	-0.04	0.19
C2		1	0.99	-0.03	0.04	0.27
C3			1	-0.19	-0.12	0.11
M1				1	1.00	0.96
M2					1	0.97
M3						1

表 2.5: 文書間の類似度 (LSA 適用後)

ると言える. そのため, df の逆数, すなわち idf を tf に掛けた値 tf $*$ idf を共起行列の値とする.

tf, idf として様々な式が提案されているが, 本研究では tf として Harman[29], idf として Sparck Jones[57] で提案されている式を利用する.

$$tf_{ij} = \frac{\log_2(a_{ij}+1)}{\log_2(|T(s_i)|)}$$

$$idf_j = \log_2 \frac{|D|}{|D'|} + 1 \text{ (ただし } D' = \{s|count(s,t_j) > 0\})$$

カテゴリ抽出手法

ソフトウェアの集合からカテゴリを抽出するために，我々は関数名や変数名，型名などの識別子に着目する．原則的には，プログラマは識別子に任意の名前をつけることが可能であるが，通常，識別子にはその動作や役割に応じた名前がつけられていることが多い．例えば，“`gtk_window`” という識別子は，なんらかのウィンドウを保存するためのものと考えられ，“`gtk_window`” が現れる文の近辺はなんらかの GUI に関する操作を行っているものと考えられる．様々なコーディングスタイルについて論じた文献においても，識別子にはその実体を表す名前をつけることがソフトウェアの品質を保つ上で重要であることが指摘されている [36, 30, 49].

そこで，LSA を利用して類似した識別子を集めて，一つの概念を構成できるのではないかと考えた．例えば “`gtk_window`” や “`gtk_main`”, “`gpointer`” といった識別子が存在していれば，そのソフトは GTK ライブラリを利用しているものと考えられる．我々は，このように生成された概念を**カテゴリ**と定義する (図 2.3)．そして，もしソフトが，そのカテゴリに含まれる識別子を一つでも持っているなら，そのカテゴリに所属するものとする．図 2.3 はカテゴリ抽出過程の例である．図 2.3 では，“`cut, copy, paste`” を “エディタ” カテゴリとして，“`CWindow, CMenu`”

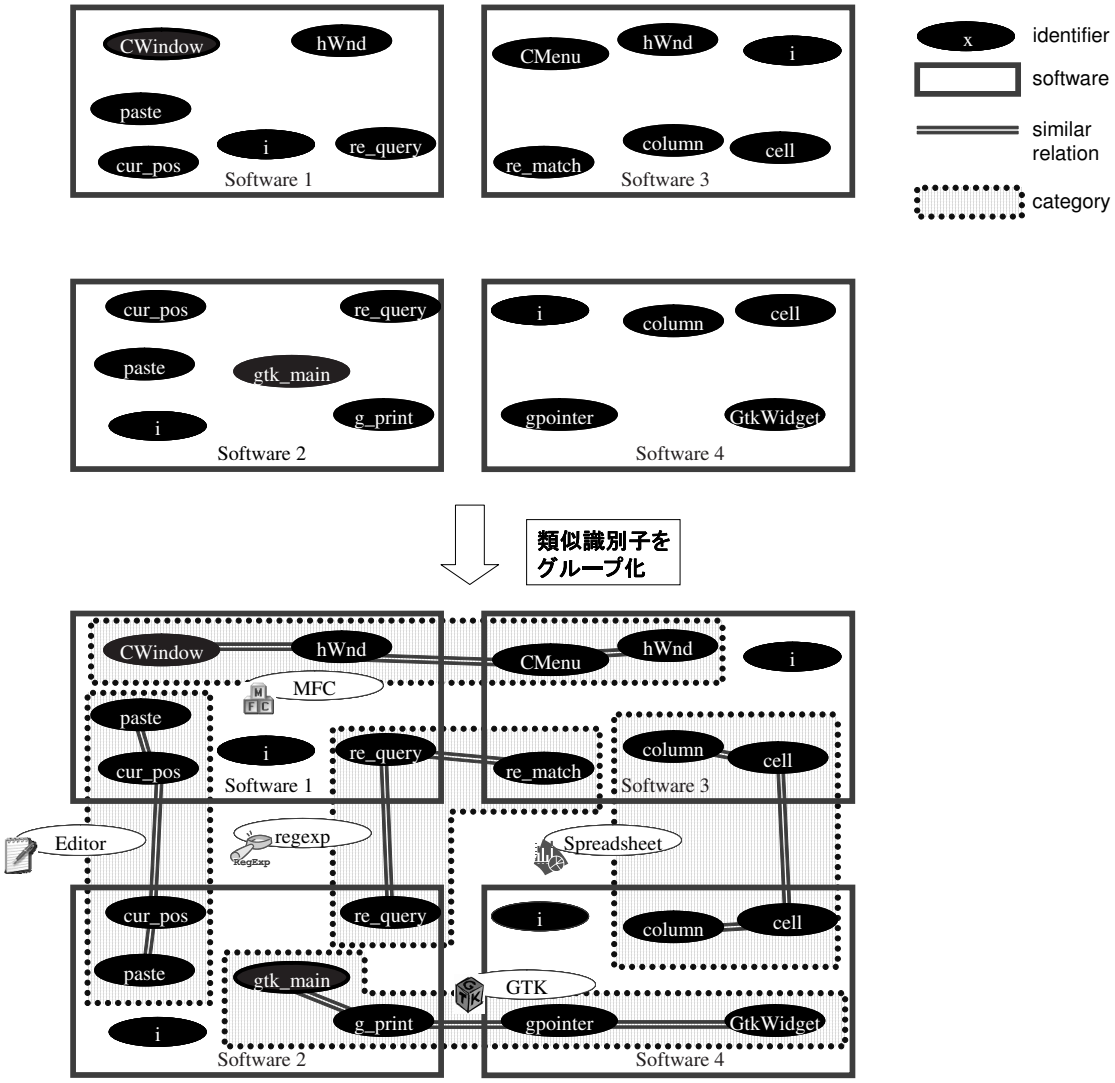


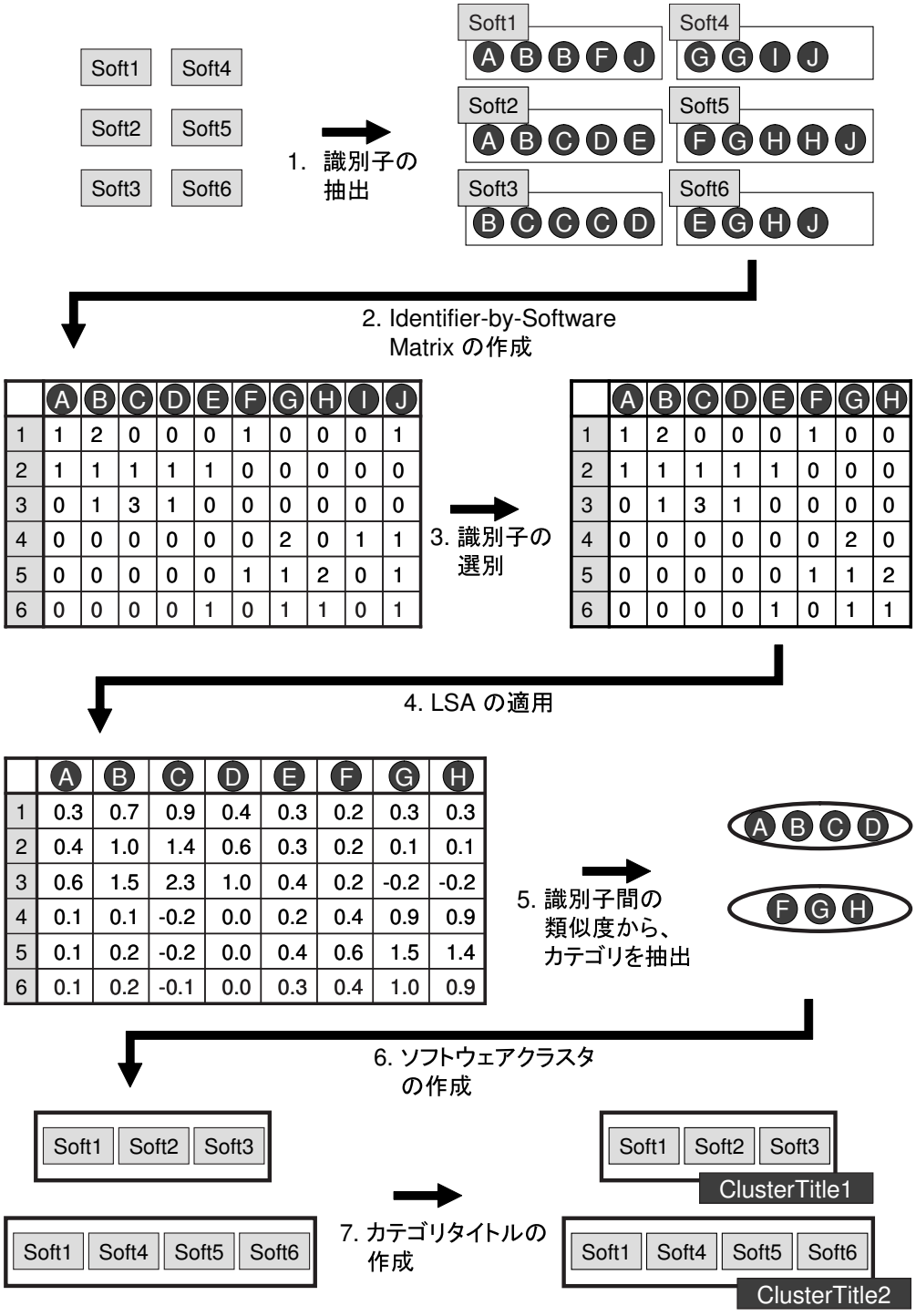
図 2.3: ソースコード内の識別子の関連からカテゴリを抽出

を “MFC” カテゴリとして抽出している．このとき，Software1 は “エディタ” カテゴリと “MFC” カテゴリに属する．

LSA は純粋に統計学的手法を用いており，事前知識の入力を一切必要としない．また LSA では意味的な関連を反映した類似度を計測できるため，類義語や同音異義語が多く含まれる状況でも，信頼性の高い類似度を得ることができる．

MUDABlue アルゴリズム

2.3.1 で述べたように，MUDABlue は類似度の高い識別子をグループ化することでカテゴリ抽出を行う．図 2.4 に MUDABlue 分類手法のデータフローを示す．本



	A	B	C	D	E	F	G	H	I	J
1	1	2	0	0	0	1	0	0	0	1
2	1	1	1	1	1	0	0	0	0	0
3	0	1	3	1	0	0	0	0	0	0
4	0	0	0	0	0	0	2	0	1	1
5	0	0	0	0	0	1	1	2	0	1
6	0	0	0	0	1	0	1	1	0	1

	A	B	C	D	E	F	G	H
1	1	2	0	0	0	1	0	0
2	1	1	1	1	1	0	0	0
3	0	1	3	1	0	0	0	0
4	0	0	0	0	0	0	2	0
5	0	0	0	0	0	1	1	2
6	0	0	0	0	1	0	1	1

	A	B	C	D	E	F	G	H
1	0.3	0.7	0.9	0.4	0.3	0.2	0.3	0.3
2	0.4	1.0	1.4	0.6	0.3	0.2	0.1	0.1
3	0.6	1.5	2.3	1.0	0.4	0.2	-0.2	-0.2
4	0.1	0.1	-0.2	0.0	0.2	0.4	0.9	0.9
5	0.1	0.2	-0.2	0.0	0.4	0.6	1.5	1.4
6	0.1	0.2	-0.1	0.0	0.3	0.4	1.0	0.9

図 2.4: カテゴリ抽出アルゴリズム

手法は 7 つの部分から構成される.

1. 識別子の抽出

まず，すべてのソフトウェアのソースコードから識別子を抽出し，これをLSA への入力として用いる．

ただし，抽出した単語から予約語，演算子はとりのぞく．これらは実装言語にのみ依存するものであり，ソフトウェアの特徴を表すものと言いがたい．また，様々なソフトウェアに均等に入っていることが予想される単語は，分析結果に悪影響を及ぼす可能性があり，計算量の観点から言ってもこのような無駄な単語を用いることは好ましくない．

またコメント中の単語も除外する．一般にコメントはソースコードの動作を比較的高い抽象度で記述していることが多い．しかし，ソフトウェアによってコメントが存在する度合やその品質にはかなりのばらつきがあり，一律に利用することが難しい．また，オープンソースプログラムなど，ライセンス条項などがコメントとして付随しているソフトウェアもあるため必ずしもコメントがプログラムの動作を表しているとは限らない．そこで，コメントも入力トークンからは除外する．

2. identifier-by-software 行列の作成

   次に，word-by-document 行列と同じように，identifier-by-software 行列を作成する．識別子が単語に，ソフトウェアが文書に相当する．

3. 識別子の選別

   LSA を適用するまえに，不要な識別子の除去を行う．本手法では，単一のソフトにのみ出現する単語，および過半数以上のソフトに出現する単語を削除する．単一のソフトに現れる単語はLSA にとって完全に不要である．また過半数以上のソフトに現れる単語は，どのようなプログラムにも普遍的に出現する分類とは無関係な単語と考えられる．

4. LSA の適用

   識別子を選別した後，identifier-by-software 行列に対してLSA を適用し，行列を変換する．LSA を適用することで，類義語が多く含まれる場合にも適切な類似度測定が可能になる．

5. 識別子間の類似度から，カテゴリを抽出

   LSA の結果得られた行列から，すべての類似度のペアについて類似度を計測する．本研究ではLSA における単語間類似度計測で一般的に使われる $cosine$ 尺度を用いる．こうして求めた類似度に基づいて，識別子群に対してクラスタ分析を適用して類似度の高い識別子同士をグループ化する．

クラスタ分析とは, 複数個の個体を, それら個体間の類似度に基づいていくつかのクラスタに分類する分析手法である. クラスタ分析にはいくつかの分析方法があるが, ここでは全てが独立したクラスタという状態から始めて, もっと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. この手法は以下のアルゴリムで表される.

(a) 初期状態 $C = \{c_j \mid 1 \leq j \leq |T|\}, c_j = \{t_j\}$

(b) $\forall i \forall j$ similarity$(c_p, c_q) \geq$ similarity(c_i, c_j) なる p, q を探索

(c) $c_p = c_p \cup c_q$, C から c_q を取りのぞく

(d) これを $|C| = 1$ となる, もしくは終了条件 $\forall i \forall j \ s \geq$ similarity(c_i, c_j), (s は与えられた閾値) を満たすまで繰りかえす.

なお, similarity(c_i, c_j) の定義方法も複数の方法が存在する. そのうちいくつかの方法は類似度の計算方法がユークリッド距離でなくてはならない. 本手法では類似度の計算方法を cos 尺度に拠っており, 本尺度は非ユークリッド距離であるためこれらの手法を利用することはできない. そのような仮定を置かない similarity(c_i, c_j) の定義法としてよく知られているものとして以下の 3 つが挙げられる.

最短距離法 二つのクラスタ間のうち, 最も距離の短い (類似度の高い) 要素間の距離をクラスタの距離とする.

$$\text{similarity}(c_i, c_j) = \cos(t_p, t_q)$$

ただし,

$$\forall x \forall y \ \cos(t_p, t_q) \geq \cos(t_x, t_y), t_p \in c_i, t_x \in c_i, t_q \in c_j, t_y \in c_j$$

最長距離法 二つのクラスタ間のうち, 最も距離の長い (類似度の低い) 要素間の距離をクラスタの距離とする.

$$\text{similarity}(c_i, c_j) = \cos(t_p, t_q)$$

ただし,

$$\forall x \forall y \ \cos(t_p, t_q) \leq \cos(t_i, t_j), t_p \in c_i, t_x \in c_i, t_q \in c_j, t_y \in c_j$$

群平均法 二つのクラスタに属する要素間の距離の平均を, そのクラスタの距離とする.

$$\text{similarity}(c_i, c_j) = \frac{\sum_{t_x \in c_i} \sum_{t_y \in c_j} \cos(t_x, t_y)}{|c_i| + |c_j|}$$

最短距離法は最も単純な方法であり, その実現も容易である. しかし, 結果として出力されるクラスタに鎖が出現する可能性が高いことが知られている. 鎖とは単一のクラスタが肥大している状態を指す. 最短距離法では, クラスタ数が多くなればなるほど他クラスタとの類似度が高くなっていく. こうして有意なクラスタ群ではなく, 巨大単一クラスタと矮小ないくつかのクラスタ, というように分割されてしまう.

群平均法は最短距離法のように鎖ができるという問題点もなく, 均等なクラスタができることが期待できる. しかし, 距離を計算するために平均を逐一計算しなおさなければならないため莫大な計算量が必要である. また, 全てのクラタ間の類似度行列を保存しておかなければならないため, クラスタの数が多くなると巨大な行列を保持しなければならない.

最長距離法は, 最短距離法と比較すると計算量が増えるが, 類似度行列を保持する必要はないため, 群平均法よりは算出にかかるコストが小さい. また, 出力されるクラスタも群平均法と遜色ない性質のクラスタが得られる. そこで本手法では最長距離法をクラスタ間の距離として用いる.

なお, この段階ではトークン数が多すぎて一度にクラスタ分析することが困難である. また, トークンの中には, いくつかのトークンと高い類似度を持って明確なクラスタを構成する分類に有用なトークンもあれば, 逆にどのトークンともそれほど高い類似度を持たず, 明確な分類の妨げとなるトークンも数多く存在する. このようなトークンを含んだまま分類を押し進めても分析結果に悪影響を及ぼす. そこで, 二段階に分けてクラスタ分析を行うことにする. 一度目のクラスタ分析において, 明確なクラスタを形成するトークンを抽出し, 二度目のクラスタ分析で, トークンのクラスタを算出する.

トークン集合 T に対して終了条件の閾値を p_{s1} として一度目のクラスタ分析した結果得られるクラスタ集合 C は次の式で表される.

$$C = \{c_1, \ldots, c_{p_{t1}} \mid \forall i \forall j c_i \cap c_j = \emptyset, c_i \subset T\}$$

このクラスタ分析で, 一定数 p_t 以上のトークンを持つクラスタに含まれるトークンのみをこの先の分析に利用する. これを有意トークン集合 T' と呼ぶ. T' は以下の式で表される.

$$
\begin{aligned}
T' &= \bigcup_{c_i \in C} c_i \\
C &= \{c_i \mid |c_i| > p_t\}
\end{aligned}
$$

次に有意トークン集合 T' のみを対象として再びクラスタ分析を行う. このように, 不要なトークンを排除してからクラスタ分析を行うことによって, より明示的なクラスタを得ることができる. 有意トークン集合 T' に対して終了条件の閾値を p_{s2} としてクラスタ分析した結果のクラスタ集合を C' とすると

$$
C' = \{c'_1, \ldots, c'_{p_{t2}} \mid \forall i \forall j c'_i \cap c'_j = \emptyset, c'_i \subset T'\}
$$

ここで得られる識別子のグループを “識別子クラスタ” と呼ぶ．**この識別子クラスタをカテゴリとする．**

6. ソフトウェアクラスタの作成

   識別子クラスタから対応するソフトウェアクラスタを作成する．ソフトウェアクラスタは，カテゴリに所属するソフトの集合を表す．ソフトウェアクラスタは，識別子クラスタに属する識別子のうち，どれか一つでも持っているソフトの集合である．

7. カテゴリタイトルの作成

   この段階で，カテゴリと各カテゴリに属するソフトの集合が得られた．最後に各カテゴリの特長を表すタイトルを作成する．

   タイトルの作成のため，ソフトウェアクラスタに属するソフトのソフトウェアベクトルを抜きだし，それを合計する．そして，そのベクトル内で値の大きい識別子を 10 個取りだし，それをタイトルとする．

2.3.2 カテゴリ描画手法

ソフトウェアリポジトリは一般に大規模であり，得られるカテゴリ数もそれだけ多くなるため，得られたカテゴリをわかりやすくユーザに提示する必要がある．MUDABlue はソフトウェアリポジトリの閲覧，検索のために (1) キーワード検索, (2) カテゴリツリー，(3)Unifiable Cluster Map (UCM) の 3 つのビューを提供する．

カテゴリツリーは後述するカテゴリーの階層構造を直接表現する部分である．もしユーザが目的とするカテゴリを知っているのなら，カテゴリツリーを用いることで直接目的とするカテゴリーにアクセスすることができる．

UCM はソフトウェアリポジトリ全体を俯瞰するのに用いられる．UCM はカテゴリとソフトウェアシステムの関連をグラフィカルに表示する．UCM はカテゴリ全体を描画することで，リポジトリ内にどのようなソフトウェアが存在するかを理解しやすくする．

キーワード検索

キーワードを用いた検索は最も一般的な検索手法である．MUDABlue ではキーワードを用いて，カテゴリおよびソフトを検索可能である．カテゴリ検索時には，カテゴリタイトル，および対応する識別子クラスタに入力されたキーワードと部分一致するカテゴリを表示する．またソフト検索時にはソフト名，もしくはソフト内の識別子とキーワードが部分一致するソフトを表示する．

カテゴリツリー

カテゴリツリーでは抽出したカテゴリの一覧を階層構造にして表示する．カテゴリの階層構造はカテゴリに共通するソフトの数を元に決定する．そのため類似したカテゴリが近くに配置され，それだけユーザにとってカテゴリの閲覧が容易になる．

本論文では，カテゴリ間の類似度として**オッズ比**を少し変更したものを用いる．母集合 S とその部分集合 $A, B \subset S$ があったとき，カテゴリ間の類似度 $or'(A, B)$ は $or'(A, B) = \begin{cases} ad/bc & \text{if } bc \neq 0 \\ ad|S|^2 & \text{otherwise} \end{cases}$ となる．ただし，$a = |\bar{A} \cap \bar{B}|, b = |A \cap \bar{B}|, c = |\bar{A} \cap B|, d = |A \cap B|$ とする．我々は $or'(A, B)$ を用いてクラスタ分析を行い，その結果をカテゴリツリーとして表示する．

Unifiable Cluster Map

MUDABlue で抽出したカテゴリの描画には，(1) 複数カテゴリへの所属を効率よく描画できること，(2) 十分なスケーラビリティを持っていること，以上 2 点の要求を満たす描画方法が求められる．このような要求を踏まえ，我々はカテゴリ描画手法として **Unifiable Cluster Map** を定義した．Unifiable Cluster Map は Cluster Map [22] を元にした手法である．

Cluster Map では，図 2.5 の左側のような関係は，右側のグラフとして表す．カテゴリは黒い丸であらわし，その丸からのびる辺によって所属しているアイテムを示す．例えばカテゴリ C にはアイテム 3, 4, 5, 6 が属しているため，それらの間に辺を引く．また，アイテム 4, 5, 6 はカテゴリ B にも属するため，カテゴリ B と

カテゴリ	アイテム
A	1, 2, 3, 7, 8, 9
B	1, 2, 3, 4, 5, 6, 10, 11
C	3, 4, 5, 6
D	7, 8, 9, 10, 11
E	7, 8, 9, 10, 11, 12

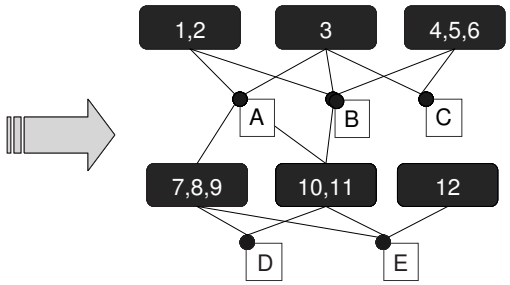


図 2.5: Cluster Map の例

アイテム 4, 5, 6 の間にも辺を作成する．スペースの節約のため，所属するカテゴリが完全に同一であるアイテム同士は一つの四角でまとめる．Cluster Map はこのような形で複数に所属するアイテムの関連を効率的に描画する．しかし，もしカテゴリが数十個を越える規模になるとノードが画面を埋めつくしてしまい，理解困難な状態に陥いる．

そこで，我々は「カテゴリの融合・展開」という操作を追加した Unifiable Cluster Map を定義，実装した [35]．UCM ではカテゴリを融合させることで，一度に表示されるノードを減らすことができ，画面がノードで埋めつくされてしまうことを防いでいる．ユーザは融合したカテゴリを再び分離することもできる．

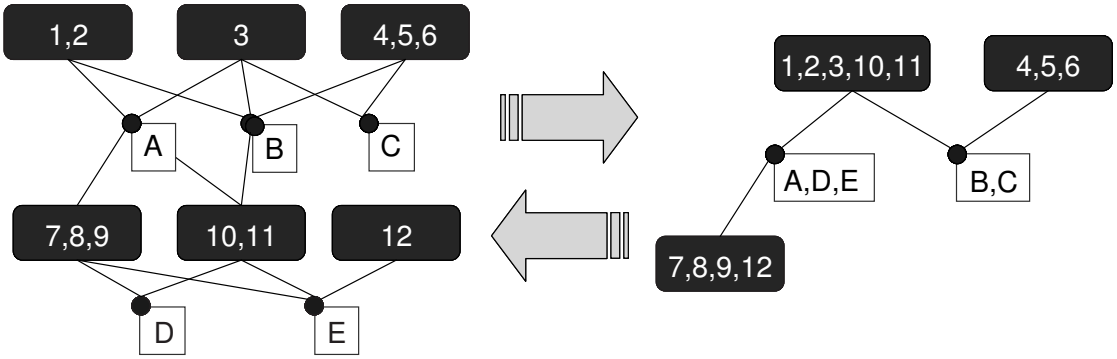


図 2.6: Unifiable Cluster Map

ソフトウェアクラスタと，結合したカテゴリが線で結ばれている場合，それらのソフトは結合しているカテゴリのうちいずれかに所属していることを示している．図 2.6 の右側は, category B と C を, category A と D と E を結合した状態を示している．

実際には，初期状態ではさきほど定義したカテゴリ階層の上位 7 個のカテゴリのみが表示され，その他のカテゴリは融合された状態で表示される．

2.4 MUDABlue システム

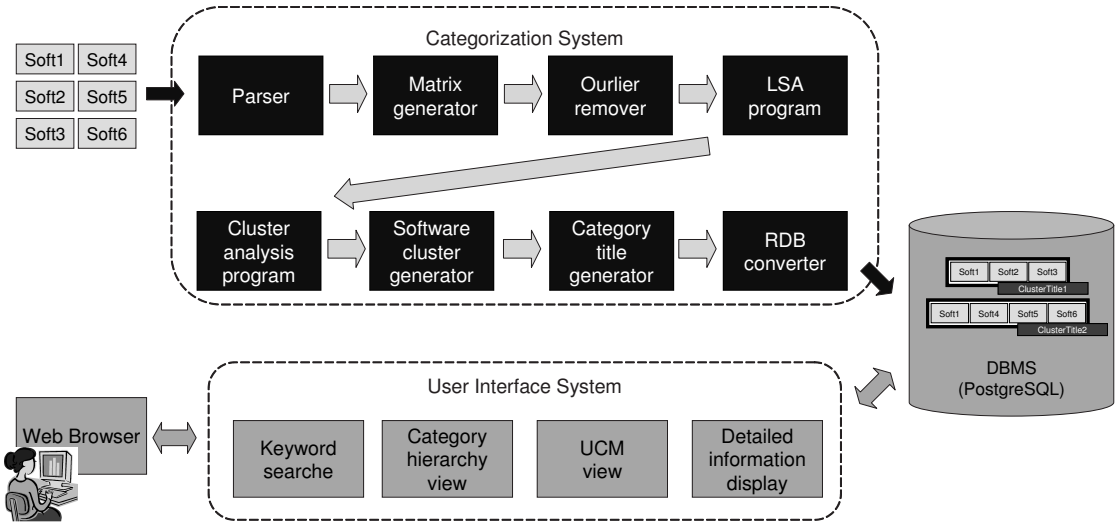


図 2.7: MUDABlue システム

本研究では自動分類アルゴリズムを用いて実際に分類を行うシステム MUDABlue を作成した．本システムは C 言語で書かれたプログラムに対応しており，与えられたプログラム集合からカテゴリの抽出と，プログラムの分類を行う．MUDABlue は Web アプリケーションとして実装されており, ユーザは Web ブラウザを通じて MUDABlue にアクセスすることでリポジトリ内のソフトウェアを検索する．

図 2.7 に MUDABlue のアーキテクチャ図を示す．MUDABlue は (1) 分類データ抽出システムと (2) ユーザインタフェースシステムから構成される．カテゴリ抽出システムは与えられたソースコードを用いてカテゴリ抽出，ソフトウェア分類を行い，データベース管理システム (DBMS) へ分類結果を登録する．ユーザインタフェースシステムは，Web インタフェースを通じて DBMS に蓄積されたデータをユーザの要求に応じて提示する．

2.4.1 分類データ抽出システム

分類データ抽出システムでは，2.3.1 節で述べたアルゴリズムに従って蓄積されたソースコードからカテゴリ抽出，ソフトウェアの分類を行い，データベースに格納する．現在，対応するデータは C 言語 のソースコードである．しかし，本手法は言語構造には依存していないため，各言語に対応するトークン切りだし部のみを作成することで，さまざまな言語に対応することが可能である．

Category	Software
boardgame	Sjeng-10.0, bingo-cards, btechmux-1.4.3, cinag-1.1.4, faile_1_4_4, gbatnav-1.0.4, gchch-1.2.1, icsDrone, libgmonopd-0.3.0, netships-1.3.1, nettoe-1.1.0, nngs-1.1.14, ttt-0.10.0
compilers	clisp-2.30, csl-4.3.0, freewrapsrc53, gbdk, gprolog-1.2.3, gsoap2, jcom223, nasm-0.98.35, pfe-0.32.56, sdcc
database	centrallix, emdros-1.1.4, firebird-1.0.0.796, gtm_V43001A, leap-1.2.6, mysql-3.23.49, postgresql-7.2.1
editor	gedit-1.120.0, gmas-1.1.0, gnotepad+-1.3.3, molasses-1.1.0, peacock-0.4
videoconversion	dv2jpg-1.1, libcu30-1.0, mjpgTools, mpegsplit-1.1.1
xterm	R6.3, R6.4

表 2.6: 実験に利用したソフトウェア

分類システムの主要コンポーネントは (1) パース部，(2) 行列生成部， (3) 不要識別子削除部， (4) LSA 部，(5) クラスタ分析部， (6) ソフトウェアクラスタ生成部， (7) カテゴリタイトル生成部， (8) データベース変換部，の 8 コンポーネントである．基本的に (1)〜(7) はアルゴリズム説明部での各手順に対応している．

パーサ部とクラスタ分析部の一部である類似度測定部は C 言語で記述されている．また LSA 部では SVDPACKC [12] を利用している．その他のコンポーネントは Perl で，またこれらコンポーネントは Unix sh スクリプトで書かれている．

2.4.2 ユーザインタフェース

ユーザ は MUDABlue システムに実装されている Web インタフェースを通じてソフトウェアシステムを閲覧，検索することができる．ユーザインタフェース部は PHP で書かれているため，サーバには PHP の実行が可能な web サーバプログラムが必要である．また，JavaScript と JavaApplet を利用しているため，クライアントには JavaScript と JavaApplett をサポートした Web ブラウザが必要である．

MUDABlue の Web インタフェースは大きく 4 つの部分から構成される．

1. キーワード検索部
2. UCM 部
3. カテゴリツリー部

4. 詳細情報表示部

.

キーワード検索部は，カテゴリまたはソフトウェアをキーワードで検索するときに利用される．UCM 部はカテゴリとソフトウェア間の関係を図式化して表示する．カテゴリツリー部はカテゴリの階層関係をそのまま表示する．そして，詳細情報表示部はキーワード検索の結果や，カテゴリ・ソフトウェアの詳細情報を表示する部分である．

MUDABlue インタフェースの各部は協調して動作する．もし UCM でカテゴリが選ばれたらカテゴリツリーでも，そのカテゴリが選択状態になる．逆もまたしかりである．

UCM を実装するために，TouchGraph [60] というグラフ描画用ライブラリを利用している．TouchGraph はノードを動的に配置する．またユーザが対話的にノードを動かすことが可能である．

2.4.3 利用例

本節では，使用例を通して MUDABlue がどのように利用できるかを示す．サンプルデータとして SourceForge から 5 つのカテゴリをランダムに選択し，その中から C で書かれた 41 個のプログラムを取得した．サンプルプログラムには計 2,663,215 行，164,102 識別子，9,519 個のファイルが存在した．実際に用いたカテゴリ，およびソフトについては表 2.6 に示す．

アプリケーションの使い道によるソフトウェア検索

もっとも典型的なソフトウェアリポジトリの使い道であるアプリケーションドメインによるキーワード検索について述べる．仮に動画編集できるアプリケーションを捜しているユーザがいるとする．このとき，ユーザが検索ウィンドウに `video` と入力し検索ボタンを押すと video という文字列をタイトル，もしくは識別子クラスタに持つカテゴリ一覧が画面に表示される．(図 2.8)

検索されたカテゴリおよび，それらカテゴリに属するソフトウェアは詳細情報表示部に表示される．図が示すとおり，動画編集に関するソフトウェアである “dv2jpg” や “libcu30”，“mjpgTools” が提示されている．

そして，ソフトウェア名をクリックすることでそのソフトウェアの詳細情報を得ることができる (図 2.9)．詳細情報画面ではソフトウェアのソースコードの他，関連するソフトウェアの一覧を得ることができる．

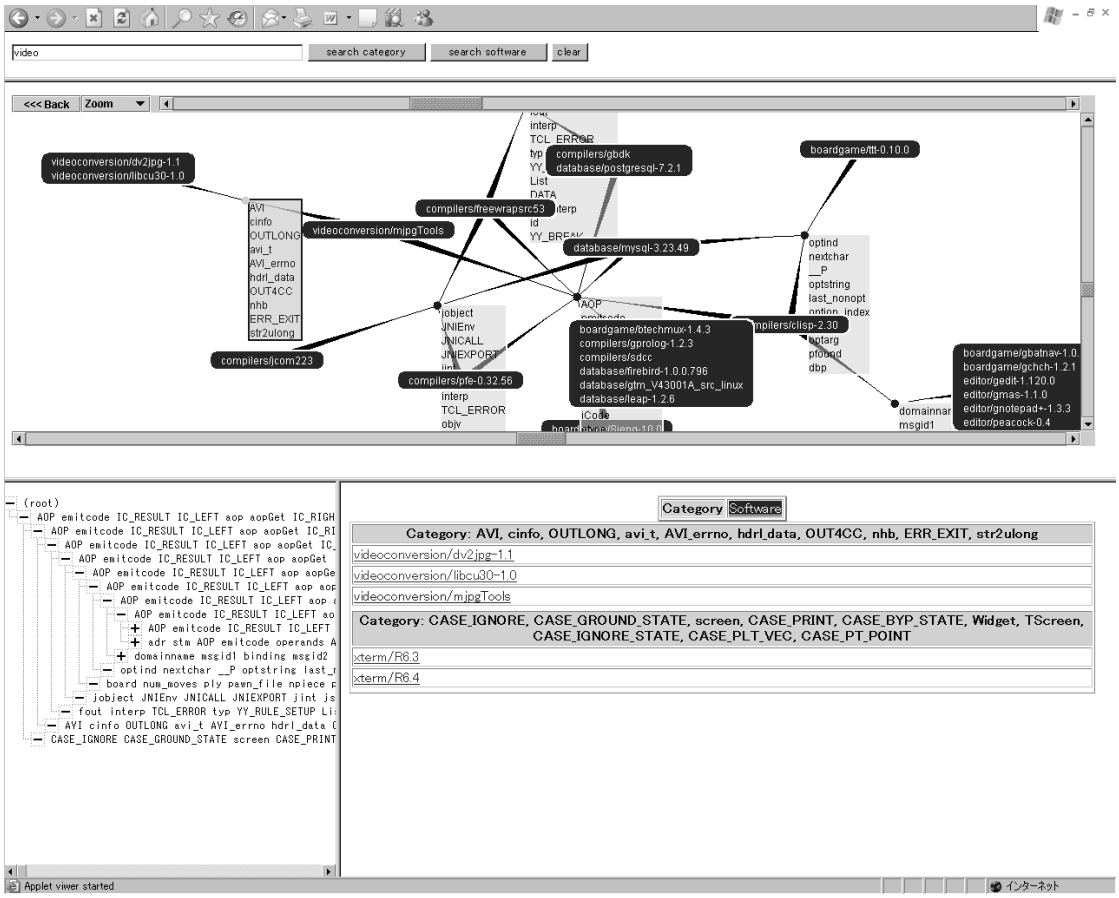


図 2.8: “video” キーワードで検索

あるソフトウェアに類似したソフトウェアの検索

またソフトウェアリポジトリの活用方法の一つに，現在開発中のプロジェクトに類似したプロジェクトを発見することが挙げられる．特に社内で運用されているソフトウェアリポジトリにおいてはこのような使われかたが特に有用である．社内で開発されている類似プロジェクトを発見することで，プロジェクト間の情報共有が促進される．

ここでは，gedit を開発している開発者がいると想定する．検索ウィンドウに gedit と入力し検索ボタンを押すと gedit という文字列をタイトルに持つカテゴリと，gedit という名前を持つソフトウェアが表示される (図 2.10).

gedit を選択すると，UCM も gedit を中心とした表示に同期する．この画面では gedit とエディタとして類似しているソフトのほか，gtk を利用しているソフトウェアが表示されており，さまざまな側面から捉えた類似性が表示されている．

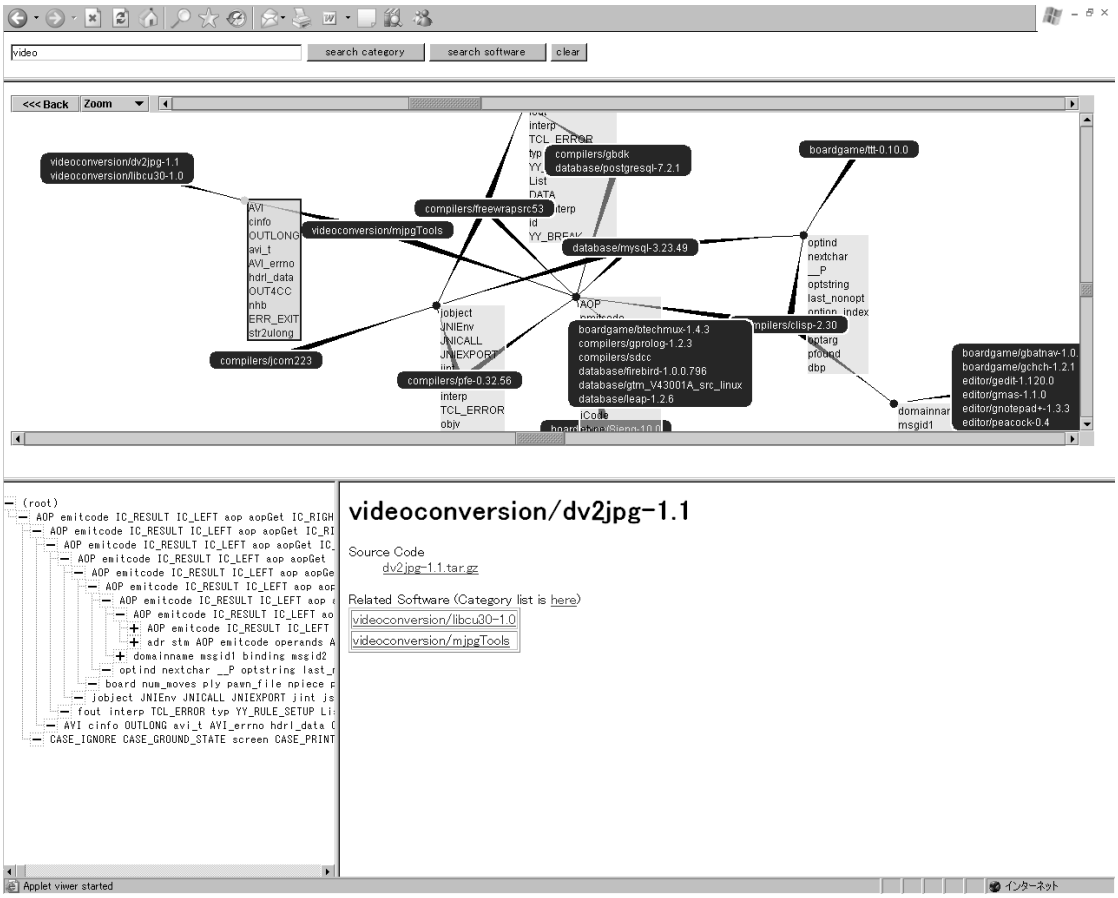


図 2.9: ソフトウェアの詳細情報

ソフトウェアリポジトリ全体の俯瞰

これまでに述べた二つの方法は適切なキーワードがあるときには有用だが，検索者がキーワードが思いつかないと検索をすることができない．このような状況ではあらかじめカテゴリ名が表示されているディレクトリ型の検索が有効である．MDUABlue では UCM という形でディレクトリ型の検索を可能にしている．

図 2.11 は MUDABlue の初期表示である．この図から，video カテゴリやエディタカテゴリに類するソフトウェアか近い位置に配置されていることがわかる．エディタカテゴリをダブルクリックすることでエディタカテゴリの詳細な分類を知ることができる．このように UCM 画面を用いることでソフトウェア間の関連を横断的に把握することができる．

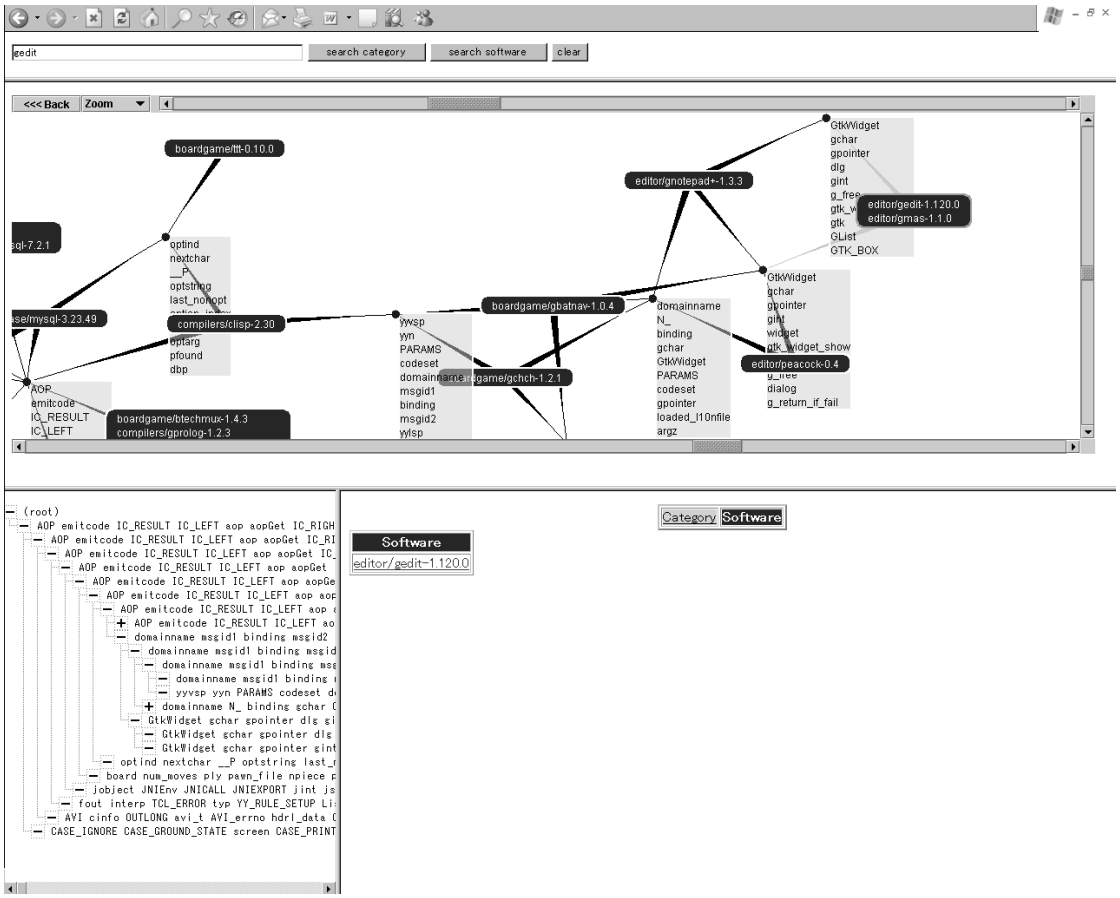


図 2.10: “gedit” キーワードで検索

2.5 実験

本論文では，(1) MUDABlue が SourceForege で定められたカテゴリをどれだけ抽出できるのか，(2) ライブラリやアーキテクチャによる分類をどれだけ抽出できるのか，という二つの観点から MUDABlue システムの評価を行った．

2.5.1 実験方法

評価用データセットとして，2.4.3 で示したものと同じ 41 個のソフトを用いた．これら評価用データセットに対して MUDABlue メソッドを適用してカテゴリ抽出を行った結果を評価する．

評価基準としては，検索アルゴリズムを評価する際によく用いられる適合率 (precision) と再現率 (recall) を用いる．本論文では各ソフトごとに，システムが関連すると提示したカテゴリが適合しているかどうかという観点で適合率と再現率の計

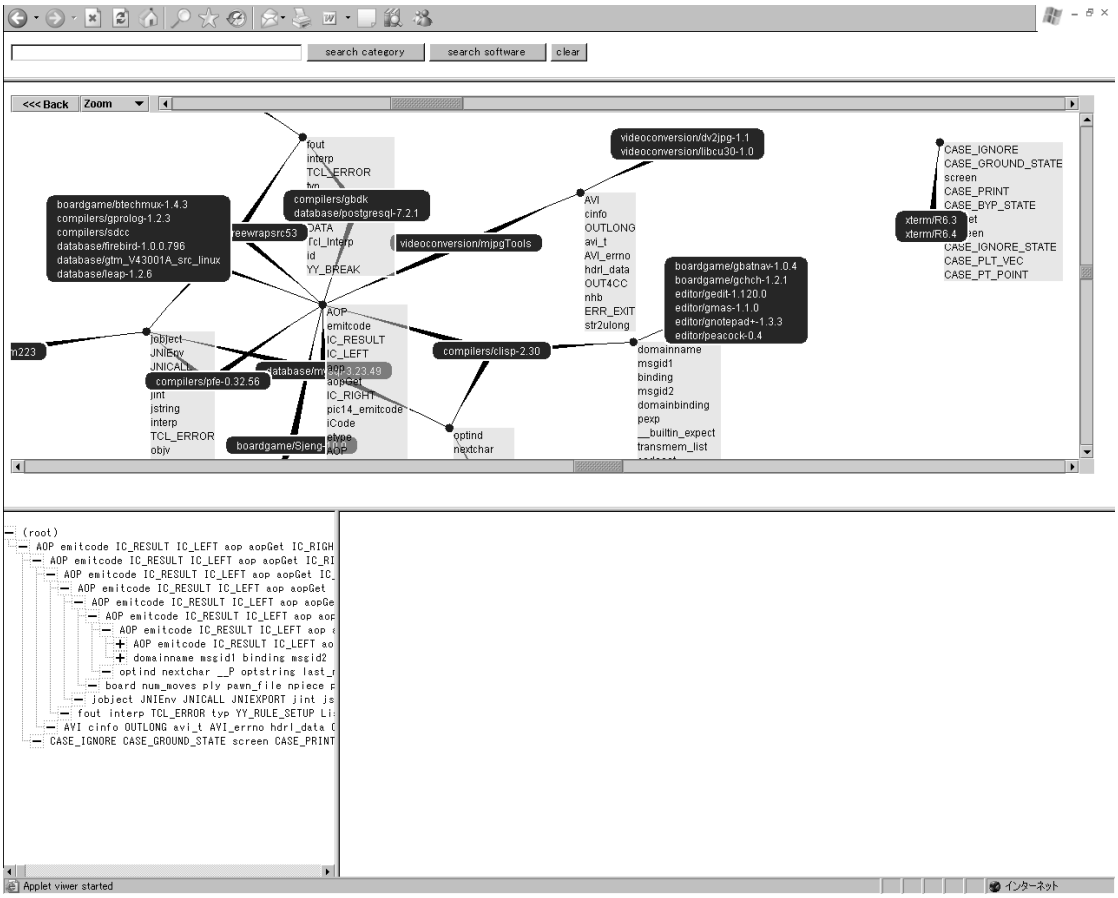


図 2.11: MDUABlue 初期画面

算を行った．具体的には, 以下の式で適合率と再現率を定義する.

$$
\begin{aligned}
\text{precision} &= \frac{\sum_{s \in S} \text{precision}_{\text{soft}}(s)}{|S|} \\
\text{precision}_{\text{soft}}(s) &= \frac{|C_{\text{MUDABlue}}(s) \cap C_{\text{Ideal}}(s)|}{|C_{\text{MUDABlue}}(s)|} \\
\text{recall} &= \frac{\sum_{s \in S} \text{recall}_{\text{soft}}(s)}{|S|} \\
\text{recall}_{\text{soft}}(s) &= \frac{|C_{\text{MUDABlue}}(s) \cap C_{\text{Ideal}}(s)|}{|C_{\text{Ideal}}(s)|},
\end{aligned}
$$

ただし, S がリポジトリ内のすべてのソフトの集合, $C_{\text{MUDABlue}}(s)$ はシステムが出力したソフト s が所属するカテゴリの集合, $C_{\text{Ideal}}(s)$ はデータセットに対して我々が定義した正解カテゴリにおいて, ソフト s が所属するカテゴリの集合である. すなわち，まず各ソフトごとに適合率，再現率を計算し，それらの平均を全体の適合率，再現率とした.

また，適合率と再現率は一般に相反する関係にあるため，これら二つの指標を統合する指標である F 値による評価も行った．F 値は，適率合と再現率の調和平均であり，適合率を p，再現率を r としたとき，

$$F = \frac{2pr}{p+r}$$

と定義される．

2.5.2 分類結果

表 2.7 に MUDABlue によって抽出されたカテゴリの一覧を示す．各行がひとつのカテゴリを表しており，各列は左からカテゴリタイトル，所属しているソフト名，対応する識別子クラスタの識別子数である．

No.	クラスタタイトル	ソフトウェア	トークン数
1	AOP, emitcode, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode, iCode, etype	compilers/gbdk, compilers/sdcc	8597
2	CASE_IGNORE, CASE_GROUND_STATE, screen, CASE_PRINT, CASE_BYP_STATE, Widget, TScreen, CASE_IGNORE_STATE, CASE_PLT_VEC, CASE_PT_POINT	xterm/R6.3, xterm/R6.4	2160
3	YY_BREAK, yyvsp, yyval, DATA, yy_current_buffer, tuple, yy_current_state, yy_c_buf_p, yy_cp, uint32	compilers/gbdk, database/mysql-3.23.49, database/postgresql-7.2.1	223
4	AVI, cinfo, OUTLONG, avi_t, AVI_errno, hdrl_data, OUT4CC, nhb, ERR_EXIT, str2ulong	videoconversion/dv2jpg-1.1, videoconversion/libcu30-1.0, videoconversion/mjpgTools	177
5	domainname, msgid1, binding, msgid2, domainbinding, pexp, __builtin_expect, transmem_list, codeset, codesetp	boardgame/gbatnav-1.0.4, boardgame/gchch-1.2.1	165

No.	クラスタタイトル	ソフトウェア	トークン数
6	board, num_moves, ply, pawn_file, npiece, pawns, moves, white_to_move, move_s, promoted	boardgame/Sjeng-10.0, boardgame/cinag-1.1.4, boardgame/faile_1_4_4	154
7	xdrs, blob, DB, UCHAR, XDR, mutex, key_length, logp, page_no, bdb	database/firebird-1.0.0.796, database/mysql-3.23.49	118
8	domainname, N_, binding, gchar, GtkWidget, PARAMS, codeset, gpointer, loaded_l10nfile, argz	boardgame/gbatnav-1.0.4, boardgame/gchch-1.2.1, editor/gnotepad+-1.3.3, editor/peacock-0.4	118
9	GtkWidget, gchar, gpointer, gint, widget, gtk_widget_show, N_, g_free, dialog, g_return_if_fail	boardgame/gbatnav-1.0.4, editor/gedit-1.120.0, editor/gmas-1.1.0, editor/gnotepad+-1.3.3, editor/peacock-0.4	104
10	AOP, emitcode, esp, IC_RESULT, IC_LEFT, obstack, aop, mov, aopGet, IC_RIGHT	compilers/clisp-2.30, compilers/gbdk, compilers/sdcc	100
11	tuple, uint32, plan, int32, lsn, elm, rec, interp, TCL_ERROR, finfo	database/mysql-3.23.49, database/postgresql-7.2.1	79
12	xdrs, blob, DB, UCHAR, XDR, mutex, key_length, logp, page_no, bdb	database/firebird-1.0.0.796, database/mysql-3.23.49	73
13	UCHAR, relation, stmt, trigger, yyvsp, yyval, t_data, plan, dbname, USHORT	database/firebird-1.0.0.796, database/postgresql-7.2.1	68

No.	クラスタタイトル	ソフトウェア	トークン数
14	fout, interp, TCL_ERROR, typ, YY_RULE_SETUP, List, DATA, Tcl_Interp, id, YY_BREAK	compilers/freewrapsrc53, compilers/gbdk, compilers/gsoap2, database/postgresql-7.2.1	50
15	GtkWidget, gchar, gpointer, dlg, gint, g_free, gtk_widget_show, gtk, GList, GTK_BOX	editor/gedit-1.120.0, editor/gmas-1.1.0, editor/gnotepad+-1.3.3	46
16	UCHAR, relation, stmt, trigger, yyvsp, yyval, t_data, plan, dbname, USHORT	database/firebird-1.0.0.796, database/postgresql-7.2.1	43
17	AOP, emitcode, mfp, ic, uchar, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT	compilers/gbdk, compilers/sdcc, database/mysql-3.23.49	36
18	adr, FX, word, stm, ED, xt, REF, prop, term, FP	compilers/gprolog-1.2.3, compilers/pfe-0.32.56	35
19	AOP, emitcode, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode, iCode, etype	compilers/gbdk, compilers/sdcc, database/firebird-1.0.0.796	31
20	dyn, FPRINTF, process_id, p_offset, ctl, rab, que, io_ptr, prior, PRINTF	database/firebird-1.0.0.796, database/gtm_V43001A	29
21	dyn, FPRINTF, process_id, p_offset, ctl, rab, que, io_ptr, prior, PRINTF	database/firebird-1.0.0.796, database/gtm_V43001A	27
22	regparse, dbp, mech, reginput, flagp, NOTHING, tuple, db, __P, regnode	boardgame/btechmux-1.4.3, database/leap-1.2.6, database/mysql-3.23.49	26
23	rectype, argp, rec, fileid, save_errno, data_len, qp, argpp, int4, dbp	database/gtm_V43001A, database/mysql-3.23.49	26

No.	クラスタタイトル	ソフトウェア	トークン数
24	AOP, emitcode, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode, iCode, etype	compilers/gbdk, compilers/sdcc, videoconversion/mjpgTools	26
25	jobject, JNIEnv, JNICALL, JNIEXPORT, jint, jstring, interp, TCL_ERROR, objv, TCL_OK	compilers/freewrapsrc53, compilers/jcom223, compilers/pfe-0.32.56, database/mysql-3.23.49	24
26	entrypoint, USHORT, TEXT, yyvsp, raddr, R, UCHAR, yyval, blob, REQ	compilers/clisp-2.30, database/firebird-1.0.0.796	17
27	int32_t, dbp, cinfo, net, unpack, argp, sinfo, cur1, purpose, mysql	database/mysql-3.23.49, videoconversion/mjpgTools	17
28	AOP, emitcode, mfp, ic, uchar, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT	compilers/gbdk, compilers/sdcc, database/mysql-3.23.49	16
29	USHORT, UCHAR, blob, REQ, NULL_PTR, hIcon, SCHAR, interp, wndclass, bdb	compilers/freewrapsrc53, database/firebird-1.0.0.796	16
30	optind, nextchar, __P, optstring, last_nonopt, option_index, uchar, optarg, pfound, dbp	boardgame/ttt-0.10.0, compilers/clisp-2.30, database/mysql-3.23.49	15
31	int4, ctl, tn, rec, semid, blkno, ti, oprtype, save_errno, AH	database/gtm_V43001A, database/postgresql-7.2.1	14
32	notify, mech, PyObject, fargs, Node, Name, pset, zone, tprintf, NOTHING	boardgame/btechmux-1.4.3, database/postgresql-7.2.1	11
33	interp, notify, dbp, tuple, mech, PyObject, uint32, plan, int32, buff	boardgame/btechmux-1.4.3, database/mysql-3.23.49, database/postgresql-7.2.1	10

No.	クラスタタイトル	ソフトウェア	トークン数
34	adr, stm, AOP, emitcode, operands, ASSERT, IC_RESULT, pred, lg, REF	compilers/gprolog-1.2.3, compilers/sdcc	9
35	yyvsp, yyn, PARAMS, codeset, domainname, msgid1, binding, msgid2, yylsp, domainbinding	boardgame/gbatnav-1.0.4, boardgame/gchch-1.2.1, compilers/clisp-2.30	9
36	ERREXIT, picture, pool_id, USHORT, get_buffer, output_buf, cinfo, xxx, UCHAR, streams	database/firebird-1.0.0.796, videoconversion/mjpgTools	9
37	REF, dyn, USHORT, vec, path_name, clause, STATUS, E, UCHAR, CSB	compilers/gprolog-1.2.3, database/firebird-1.0.0.796	8
38	AOP, emitcode, pfile, ic, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode	compilers/gbdk, compilers/sdcc, database/postgresql-7.2.1	7
39	ic, ply, npiece, score, AOP, pawn_file, uchar, bking_loc, wking_loc, emitcode	boardgame/Sjeng-10.0, compilers/gbdk	7
40	clause, cinfo, pred, ci, Group, Np, word, X, A, tmp4	compilers/gprolog-1.2.3, database/postgresql-7.2.1, videoconversion/mjpgTools	6

表 2.7: 41 ソフトウェアの全分類結果

この実験では合計 40 個のカテゴリが得られた．そのうち 18 個は SourceForge で決められた用途による分類に合致するものであり，11 個はライブラリやアーキテクチャに基づく新しく抽出されたカテゴリであった．残り 11 個はそのどちらでもなく，意味のないカテゴリであった．用途による分類となるカテゴリは No．1, 2，4，5，6，7，10, 11, 12, 13, 15, 16, 18, 20, 21, 23, 31, 34 であり，新しいカテゴリは No.3 (YACC カテゴリ)，No.8, 9 (GTK カテゴリ)，No.14 (SSL カテゴリ)，No.22 (regexp カテゴリ)，No.25 (JNI カテゴリ)，No.29, 33, 40 (Win32 API カテゴリ)，No.30 (getopt カテゴリ)，No.32 (Python/C カテゴリ) である．

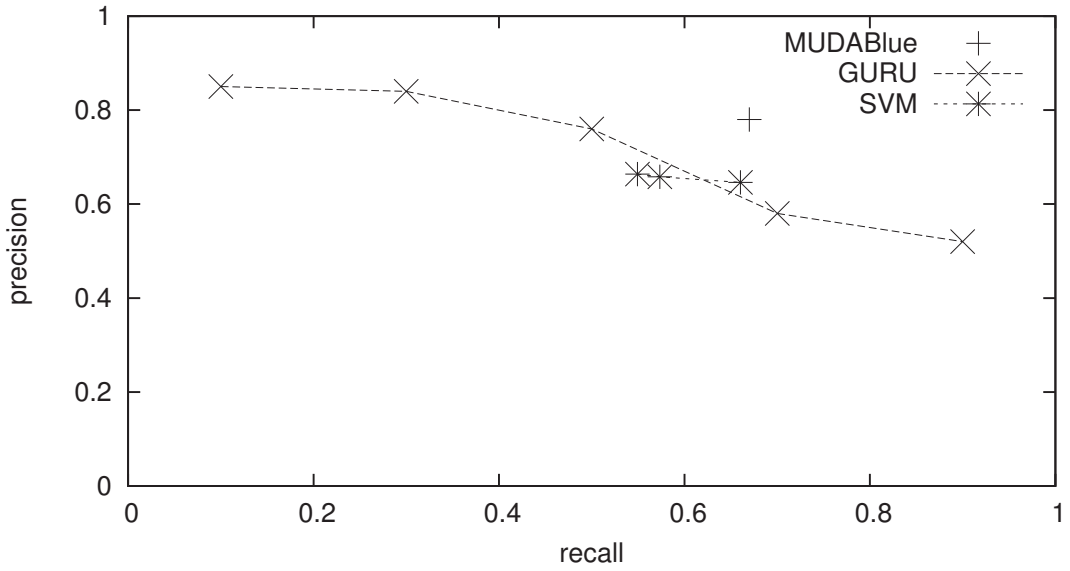


図 2.12: 適合率-再現率

図 2.12 に適合率と再現率の結果を示す．比較のため，Maarek らが提案している GURU [45] と SVM[2] を利用した分類手法である Ugurel ら [61] の適合率・再現率もあわせて示す.

これらの研究は MUDABlue と同様，Information Retrieval(IR) 手法を利用したソフトウェア分類手法である．GURU はベクトル空間モデルを利用してソフトウェア間の類似度を定義することで分類を行っている．この研究では Unix ツール群 (ls，mkdir 等) を分類する実験を行っており，その際には入力として各ツールの man ページを用いている．Ugurel らの研究では，各ソフトウェアに付随するドキュメントに対して SVM を適用し，ソフトウェアの用途による分類を試みている．また，ソースコードに対しても SVM を適用しているが，こちらについては実装言語の推定にとどまっており，本実験との関連は薄いため，前者のソフトウェアの用途による分類の結果とのみ比較する．

図 2.12 から，MUDABlue はこれらの研究に対して同程度の適合率，再現率となっていることがわかる．実際に MUDABlue の結果の F 値が 0.72 であるのに対して，GURU の結果のうち F 値の最大値は 0.6591 であり (適合率 0.9，再現率 0.52 のとき)，Ugurel らの結果の F 値の最大値は 0.6531 である (適合率 0.66，再現率 0.65 のとき).

GURU や Ugurel らの結果はそれぞれの論文から直接引用したものであり，これらの結果は異なるデータセットに対して適用した得られた結果である．そのため

[2]SVM: Support Vector Machine

この数値だけで優劣を決定することはできない．しかしこの結果から，本研究はこれら過去の研究とは大きく異なったアプローチで分類を行っているが，同程度の信頼性を保っていることがわかる．

2.6 考察

2.6.1 カテゴリ抽出手法

本論文では，MUDABlue が用途による分類にあわせて，ライブラリによる分類も適切に行なえることを示した．実験において，MUDABlue は GTK や yacc カテゴリなどを自動的に抽出している．MUDABlue は人による知識入力が必要ないため，新しいライブラリを使うソフト群が追加されても，これを自動的に抽出することが期待できる．

さきほどの実験では，適合率，再現率において MUDABlue は既存の研究に比べて同程度，もしくはそれらを上まわる値を示した．さらに，MUDABlue は既存の研究と比較して依存する情報が少ないという利点がある．まず MUDABlue ではカテゴリの抽出も自動的におこなうため，カテゴリ集合を事前に定義する必要はない．また MUDABlue はソースコードを解析して分類を行うためソースコードのみで分類を実行できるが，既存の手法はソフトウェアに付随するマニュアル，ドキュメントを解析している．第 2.2 節でも触れたとおり，付随するドキュメントの量や質はソフトウェアによってばらつきが大きく，また更新されておらず実装と異なる記述となっている場合も多い．そのため，ドキュメントではなく，ソースコードを解析するほうが望ましい．

また，既存の分類では一つのソフトウェアに対して単一のカテゴリを割りあてるが，MUDABlue では一つのソフトウェアに複数のカテゴリを割りあてることを許している．この特性は大規模なソフトウェアリポジトリを分類する際には必要なものであり，既存の研究との大きな違いである．

導出カテゴリの粒度

ただし，MUDABlue が導出するカテゴリは，全体として細かいクラスタになる傾向がある．2.5 での実験では一つのクラスタに含まれるソフト数の平均値は 2.6，最大値が 5，最小値と最頻値がともに 2 となっている．

このように細かい分類となっているのは，大きな単位でクラスタを切りだしてしまうと著しく精度が下るからである．この現象を克服するためには不要識別子

のフィルタリングをより工夫して，分類をする上でノイズとなる識別子を排除する必要がある．

カテゴリタイトル

MUDABlue では各トークンクラスタの上位 10 個を抽出して，それをクラスタのタイトルとしているが，カテゴリタイトルも解釈の難しいタイトルが生成される場合がある．カテゴリ No.4, 6 は「“AVI”, “ply”」等，比較的理解しやすいタイトルの生成に成功しているが，カテゴリ No.1 などを見て直感的にどのようなソフトウェアが分類されているかを理解することは困難といえる．

一般にライブラリやアーキテクチャに関するクラスタについては，理解しやすいカテゴリタイトルが，ソフトウェアの使い道による分類に関するクラスタについては理解しにくいカテゴリタイトルが生成される傾向にある．これは識別子が使い道の関連する概念が使われることはあっても，使い道の名前そのものが使われることが少ないからではないか，と考えている．例えばエディタに関するソフトウェアであれば，“copy_region”, “search_by_re” というような識別子が表われることがあっても，“edit” という識別子が出現することは，ほぼなかった．

この問題に対処するには，ソースコードだけでなくドキュメントも入力として利用することが考えられる．また，大規模なコーパスを前もって作成してもらう，またはタイトルを人に直接入力してもらう等，人手をかける方法も考えられる．ただし，どちらの方法も自動的に分類を行うという MUDABlue の利点を減じてまうため，極力労力が少なくてすむような実装にする必要がある．

2.6.2 MUDABlue インタフェース

第 2.4.3 節で触れたように，MUDABlue はソフトウェアリポジトリの閲覧，検索のためにいくつかのインタフェースを備えている．MUDABlue はキーワード型検索とディレクトリ型検索の両方を実装しており，1) 特定ソフトウェアの検索，2) カテゴリによる検索，3) リポジトリ全体の閲覧という 3 種のシナリオにおいて有効に活用できることを示した．

複数の閲覧検索手法を組みあわせることで，MUDABlue はより実用的な検索を実現している．Frakes ら [23] はいくつかの閲覧手法の比較を実証的に行なっている．その結果，全体的には検索手法ごとの適合率，再現率には有意差が認められないこと，しかし検索手法によって実際に提示されるアイテムは異なると結論づけている．そのため，本システムでは両タイプの検索手法を組みあわせることで，効率的な閲覧を可能にしている．

Unifiable Cluster Map

非排他的集合の関係を描画するために，我々は Cluster Map を元に Unifiable Cluster Map を実装した．Cluster Map はベン図から派生した手法の一つであるが，そのほかにも InfoCrystal [58] もそのような手法の一つである．InfoCrystal では，ベン図での各部分集合 ($A \cup B \cup \overline{C}$ や，$\overline{A} \cup B \cup \overline{C}$ 等) を切りはなし，それぞれをアイコン化する．各アイコンには，その条件を満たす要素が何個あるかを示す数字がつけられる．InfoCrystal は部分集合を切りはなして描画することで，集合数の増加に対応している．しかし，InfoCrystal では描画する集合が n 個あると，アイコンを配置する枠が n 角形になるため，集合数が大きくなるにつれて視認性が落ちる．また，InfoCrystal はカテゴリを融合させて描画するのも困難である．

非排他的集合を描画する方法として，各アイテムを所属する集合に応じて何らかの空間にマッピングする手法も存在する．酒井ら [53] や Allan ら [1] はそのような手法の一つである．これらの研究では，同じ集合に属する要素は近くに，そうでない要素を遠くに配置することで，要素間の関係を表現している．しかし，これらの手法ではカテゴリは制約条件として用いられるだけで，カテゴリそのものの描画は行われない．

カテゴリ描画手法

2.4.3 での利用例を通じて，MUDABlue が複数の閲覧検索手法を組みあわせて，より実用的な検索を実現していることを示した．Frakes ら [23] はいくつかの閲覧手法の比較を実証的に行なっている．その結果，全体的には検索手法ごとの適合率，再現率には有意差が認められないこと，しかし検索手法によって実際に提示されるアイテムは異なると結論づけている．そのため，本システムでは両タイプの検索手法を組みあわせることで，効率的な閲覧を可能にしている．

非排他的集合の関係を描画するために，我々は Cluster Map を元に Unifiable Cluster Map を実装した．そのほかにも InfoCrystal [58] もそのような手法の一つである．また，酒井ら [53] や Allan ら [1] の手法のように，各アイテムを所属する集合に応じて何らかの空間にマッピングする手法も存在する．

2.7 関連研究

まず，ソフトウェアの自動分類を行う研究としては，実験の節でとりあげた Maarek らの GURU [45] や Ugurel ら [61] の手法のほかに，山本らの SMMT [63] が挙げられる. SMMT は二つのソフトウェアシステムの類似度を計測する. SMMT ではコー

ドクローン検出技法 [33] に基づいてソースコード行単位での類似度を測定する. 山本らは SMMT を用いて 4.4BSDLite から派生したソフトウェアである FreeBSD, NetBSD, OpenBSD の各バージョン間の類似度を計測し, これらのソフトウェアを適切に分類できることを示している.

しかし, SMMT での類似度の定義は二つのソフトウェア全ての行のうち, コードクローンを含む行, および全く同一の行が存在する割合としている. そのため, 同じソースから派生したようなソフトウェア間の類似度は適正な値が出力されるが, 全く出自が異なるソフトウェア間では類似度の値が極端に低くなる.

IR 手法を用いてソースコードから何らかの有用な情報を抽出する研究として, ソフトウェアクラスタリングに関する研究がある．ソフトウェアクラスタリングとは，ソフトウェアの理解支援のために，一つのソフトウェアを機能単位で分割する手法である．ソフトウェアクラスタリングの研究については，LSA を使う研究 [46] のほか，自己同一化マップを利用する研究 [15]，ファイル構成やファイル名を利用する研究 [2]，コールグラフ等のプログラムの構成に基づく研究 [16, 43] などがある．これらの研究では，ファイルや関数などのコンポーネントを対象に IR 手法を用いて類似度を計測する．その結果に基づいてコンポーネントを分類することでソフトウェアクラスタリングを実現している.

また，再利用可能なコンポーネントを取得するために，IR 手法を用いる手法も提案されている．CodeBroker [64] は Java に対応したコンポーネント取得システムの一つである．CodeBroker は開発者がソースコードに記述した JavaDoc を自動的に取得し，LSA を用いて類似したコメントを持つメソッドを提示する.

Marcus ら [47] はソースコードと設計書などの開発文書の間の関連を自動的に抽出する手法を提案している．ソースコード内のコメントと設計文書との類似度を計算して，ソースコードに対応する開発文書を提示したり，設計書の記述に対応する実装を提示する.

この他，SVM を利用してバグ追跡システムに新しく登録された不具合の担当者を自動的に割りあてたり [44]，プログラム中に潜在的に含まれるバグを自動的に発見する手法 [13] が提案されている.

Lucca ら [44] は SVM をバグ追跡システムに適用して，新しく到着した不具合の担当者を誰にするかを自動的に判別する手法を提案している．この手法では新しく到着した不具合の説明と，既存の不具合情報とを比較し，類似度の高い不具合の担当者を，新しい不具合の担当として提示する.

Burnl [13] は SVM と決定木を用いてバグを自動発見する手法を提案している. まず準備として，バグを含むソースコードと，バグを含まないソースコードを大量に用意する．そして，ソースコードに含まれる不変式を抽出して，学習器のテ

ストデータとする．最後にその学習器を使って未知のソースコードにバグが含まれるかどうかを判定している．

2.8 結論と課題

本章では，ソフトウェアリポジトリ自動分類システム MUDABlue の設計および評価について述べた．MUDABlue は決められたカテゴリにそってソフトウェアを分類するのではなく，カテゴリそのものもソースコードから抽出して分類を行う．そのため MUDABlue は事前の知識入力を必要としない．また，MUDABlue システムは取得したカテゴリに基づいてソフトウェアリポジトリを閲覧するインタフェースも備えている．我々がおこなった実験では，用途による分類のみならず依存ライブラリやアーキテクチャによる分類もあわせて行えることを示し，また MUDABlue の既存研究に対する優位性も示した．

MUDABlue は完全に自動化されたシステムである．自動化は大規模なソフトウェアリポジトリに対して適用する際には重要な特性であるが，現在のところ人の入力も活用することで改善が期待できる部分もある．たとえば，タイトルの理解しやすさの向上はそのような部分の一つである．

第3章　版管理システムを用いたクローン履歴抽出手法

3.1　導入

近年，ソフトウェアの大規模化にともないソフトウェアの開発工程において保守工程の占める割合は年々増加の一途を辿っている．保守工程におけるさまざまな問題のなかでも非常に大きな問題の一つとして，ソースコード中に含まれるクローンが挙げられる [33]．もしクローンの一つに不具合がみつかった場合，すべてのクローンを調査し，それぞれに対して同様の修正を施すことになる可能性が高い．この作業はソフトウェアが大規模であればあるほど困難となる．

しかし，実際には関連性が高いにも関わらず，最新のバージョンを分析するだけではクローンとして抽出できないコード片が存在する．図 3.1 はそのようなコード片の例である．図 3.1 は時刻 $t-1, t$ におけるソースコードに含まれるクローンの状態を表している．時刻 $t-1$ において一つのクローンセット (互いにクローン関係にあるクローンの集合) となっているものが，一部のクローンに編集が加えられた結果，時刻 t において二つのクローンセットに分かれている．

もしクローンセット B に不具合が見つかった場合には，クローンセット B だけではなく，若干の編集を加えられたクローンセット C もまた見過ごせないコピーの一部である (以後，クローンセット C をクローンセット B の分岐クローンセットと呼ぶ．逆も同様)．しかし，これまでに提案されているクローン分析では現時点での情報のみを用いて分析を行う．そのため，クローンセット B にとってはクローンセット C も，クローンセット A と同じく無関係なクローンセットでしかない．

このようなクローンセット同士の関係を抽出するには，過去のソースコードにおけるクローン解析結果と，過去に存在したクローンが現在のソースコードのどこかに対応しているのか，対応しているとすればそれはどこなのかという情報が必要である．過去のクローン解析結果については，1.2 節で触れた版管理システムによって管理されているソフトウェアであれば過去のソースコードを入手できるため，これにクローン解析を適用することで過去のクローン情報を得ることがで

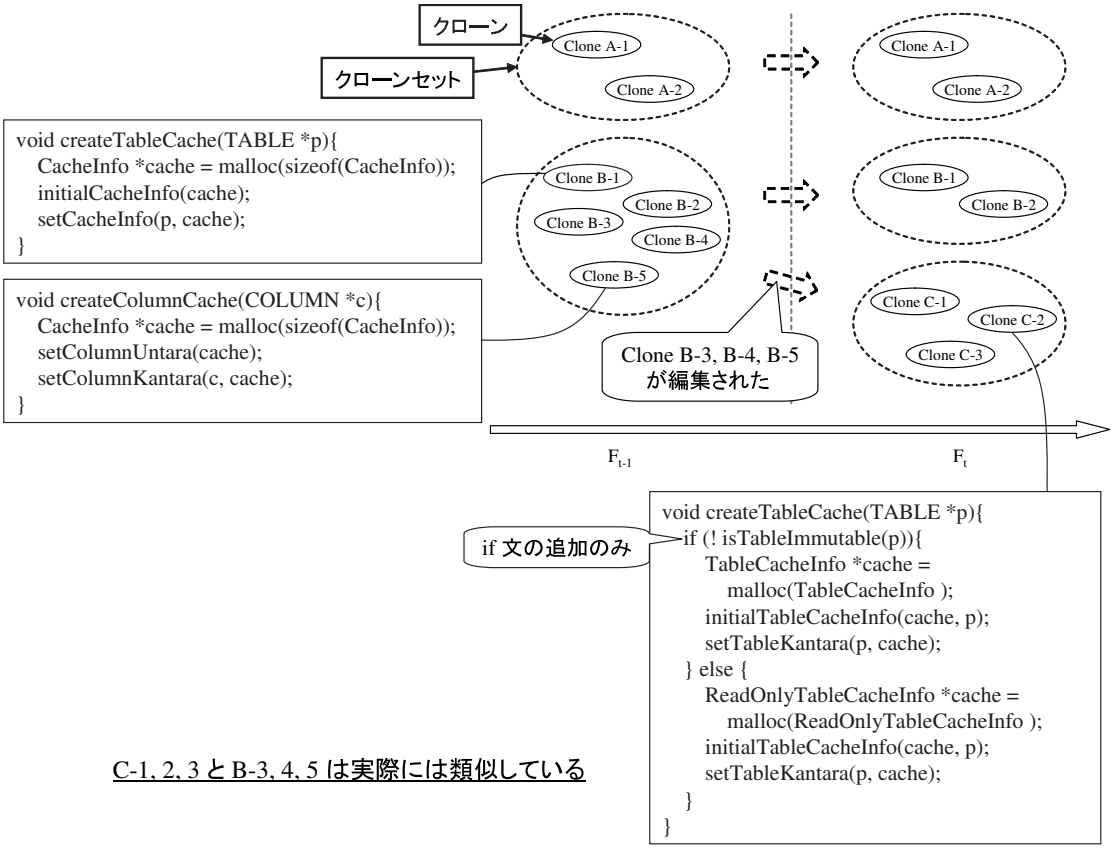


図 3.1: クローンの変更履歴

きる．しかし，後者の過去のクローンが現在のそれとどのように対応するのかを知る手段は提供されていない．

そこで，本研究では過去と現在のクローン間の繋がりを分析するための手法として，クローン履歴分析を提案する．クローン履歴分析では，現時点のクローンが過去のバージョンのどのクローンに対応するのか，今あるクローンがどのような変遷を辿ってきたのかを特定する．

また，クローン履歴分析を行う過程においては，過去から現在にいたるソースコードを連続的にクローン分析を行う．このとき，時系列にそったソフトウェアに含まれるクローンの量や割合の変化が得られる．これらの情報はソフトウェアの開発工程や開発者の技量を評価する上での重要な指標となりうる．例えば，クローンが大量にあるクローンセットについて，そのクローン履歴を調査することができれば，クローンがいつ，どのような形で増えたのかを知ることができる．また，クローンの比率のグラフを見ることで，クローンが異常に増加していないかどうかを監視することができる．

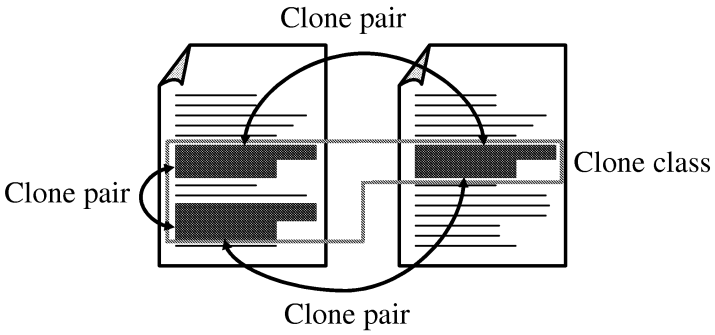


図 3.2: クローンペアとクローンクラス

3.2 クローン分析ツール CCFinder

本節では，提案手法の記述に先立ってクローン履歴分析手法で用いる CCFinder ついて，その概要を説明する．

1 章で触れたようにクローン分析手法にはさまざまな手法が存在するが，本研究では字句解析ベースの検出ツール CCFinder[33] を利用してクローン履歴の分析を行う．CCFinder は高いスケーラビリティを有しており，大規模なソフトウェアに対しても実用的な時間でクローンの抽出を行える．また，実際にさまざまな大規模ソフトウェアへ適用され，その有用性が確認されている [51].

CCFinder は，クローン分析を行う際に空白や改行，コメント等を除去するとともに，入力テキスト中の変数名や関数名等を同一記号に縮退させる．その後に，しきい値以上の長さの極大共通字句列を探索し，全ての対のリストを出力する．

CCFinder におけるクローンモデルを図 3.2 に示す．あるトークン列中に存在する 2 つの部分トークン列 α，β が等価であるとき，α は β のクローンであると定義する (その逆もクローンであるという)．また，(α，β) をクローンペアと呼ぶ．α，β それぞれを真に包含するどのようなトークン列も等価でないとき α，β を極大クローンという．また，クローンの同値類をクローンクラスと呼び，ソースコード中でのクローンを特にコードクローンと呼ぶ.

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 [31].

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．

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．

3.3 諸定義

本節では分析対象となるクローンについて定義を与える．その上で異なる時点におけるコード片の対応関係を表す写像の概要と，クローン履歴関係の定義について述べる．

3.3.1 クローン

クローン履歴分析のために定義したクローン履歴モデルを図 3.3 に示す．本モデルでは時間と共に変更されるソースコードを Δt ごとに区切った状態で考える．分

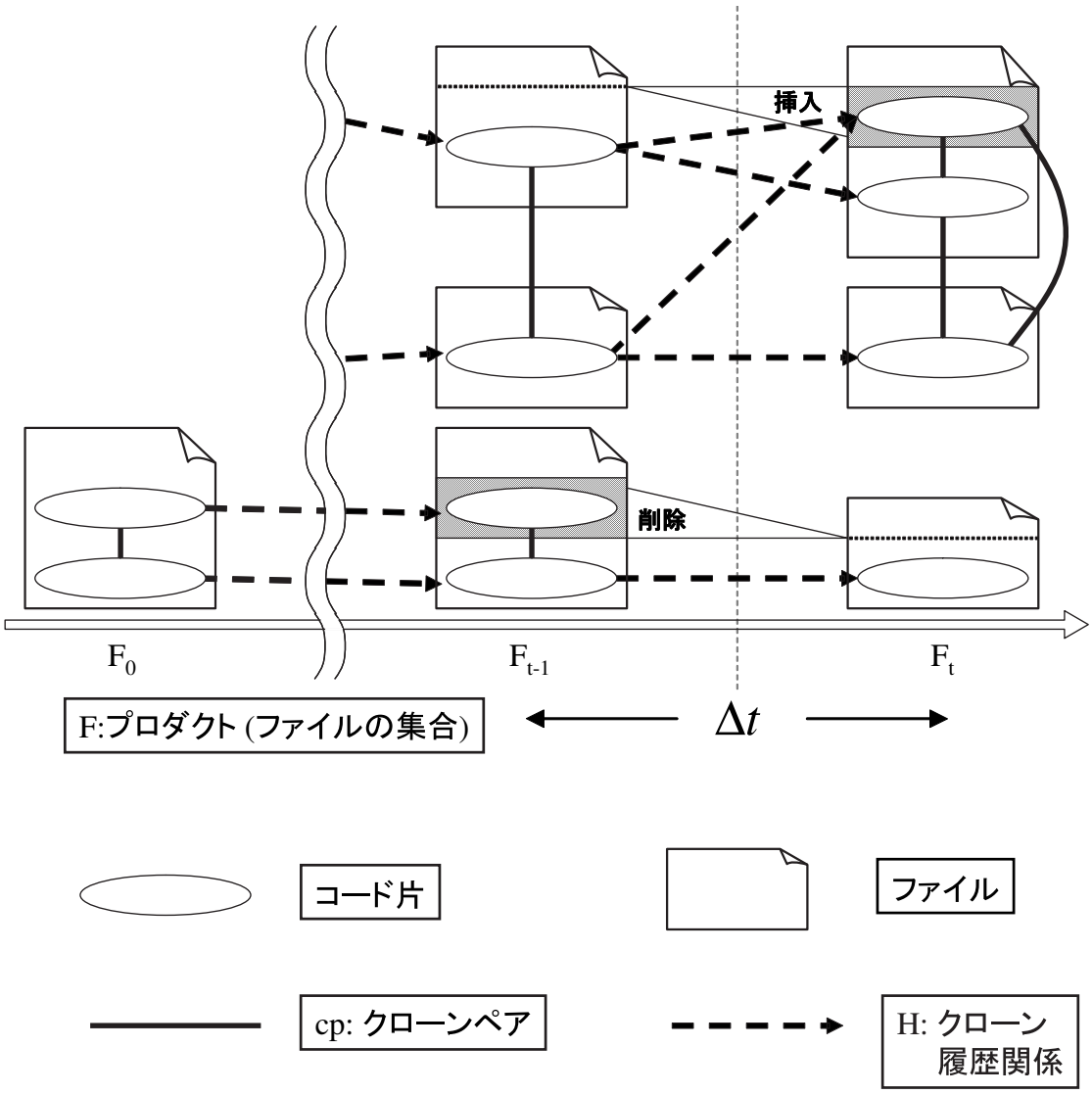


図 3.3: クローンとクローン履歴関係

析の対象を版管理システムの管理下にあるソースコードファイルの集合とし，ある時刻 t における集合を**プロダクト (Product)** F_t と呼ぶ.

いま，各ファイルを文字列と考え，その連続する部分列を**コード片 (Code snippet)** と呼ぶ．CCFinder に，F_t の各ファイルを入力として与えると，コード片の対 (a, b) の列を出力する．これを F_t に関する**クローンペア (Clone pair)**，a や b をそれぞれ F_t に関する **クローン (Clone)** と呼ぶ．このとき a と b は**クローン関係 (Clone relationship)** にあるという．また，クローン関係を同値関係と考え，その同値類を**クローンセット (Clone Set)** と呼ぶ.

3.3.2 コード片の写像

次に時刻 t より Δt だけ過去の時点におけるプロダクト F_{t-1} に含まれるコード片に対して，F_t に含まれるコード片への写像を定義する．例えば F_{t-1} にあるコード片がもし編集されていなければ，F_t にも同じ内容のコード片が必ず存在する．また，時刻 t において編集されたテキストを含むコード片についても，図 3.5 のように編集操作を考慮して近似的に対応関係にあるコード片候補を求められる．このような両者の関係を写像として定義する．本写像によって F_{t-1} のコード片 r が空文字列でない F_t のコード片 s に写像される時，r を s の**親**，s を r の**子**とする．なお写像の詳細な定義は 3.4.3 節で述べる．

3.3.3 クローン履歴関係

F_{t-1} のコード片 a と F_t のコード片 b が以下のいずれかを満たす場合，a, b 間には **クローン履歴関係 (Clone history relationship)** があるといい，クローン履歴関係が成りたつコード片の組 (a, b) の集合を H_t と表す (図 3.4)．H_t は以下に述べる HC_t, HT_t, HA_t の和集合として定義される．

1. HC_t: (a, a') が F_{t-1} に関するクローンペア，(b, b') が F_t に関するクローンペアで，b が a の，b' が a' の子となる a', b' が存在する．

2. HA_t: a は F_{t-1} に関するクローン，b は F_t に関するクローンで，かつ (a, b) は $\{F_{t-1} \cup Change_{\Delta t}\}$ に関するクローンペアである．ただし，$Change_{\Delta t}$ は F_{t-1} から F_t の変更時に編集されたコード片の集合とする．

3. HT_t: a は F_{t-1} に関するクローン，もしくは F_{t-1} のコード片で $\exists x, (x, a) \in HT_{t-1}$ ．b は a の子で a と b は類似している (類似の詳細は後述)．

全てのクローンは以下の 3 つのパターンに分類できる．すなわち，F_{t-1}, F_t の両方に存在する “継続しているクローン”，F_{t-1} のみに存在する “削除されたクローン”，そして F_t にのみ存在する “追加されたクローン” である．そして，上記 3 つの定義はそれぞれのクローンに対応する形で定義している．

HC_t は F_{t-1}, F_t の両方に存在する同一のクローンを表す．本定義は F_{t-1} でクローンペアを構成するクローン a, a' が，写像先の b, b' においてもクローンペアを構成している，ということを表す．

HA_t は F_t において追加されたクローンの履歴を表す．このようなコード片は，既にあるクローンセットのどれとも等しくクローン履歴関係があると考える．こ

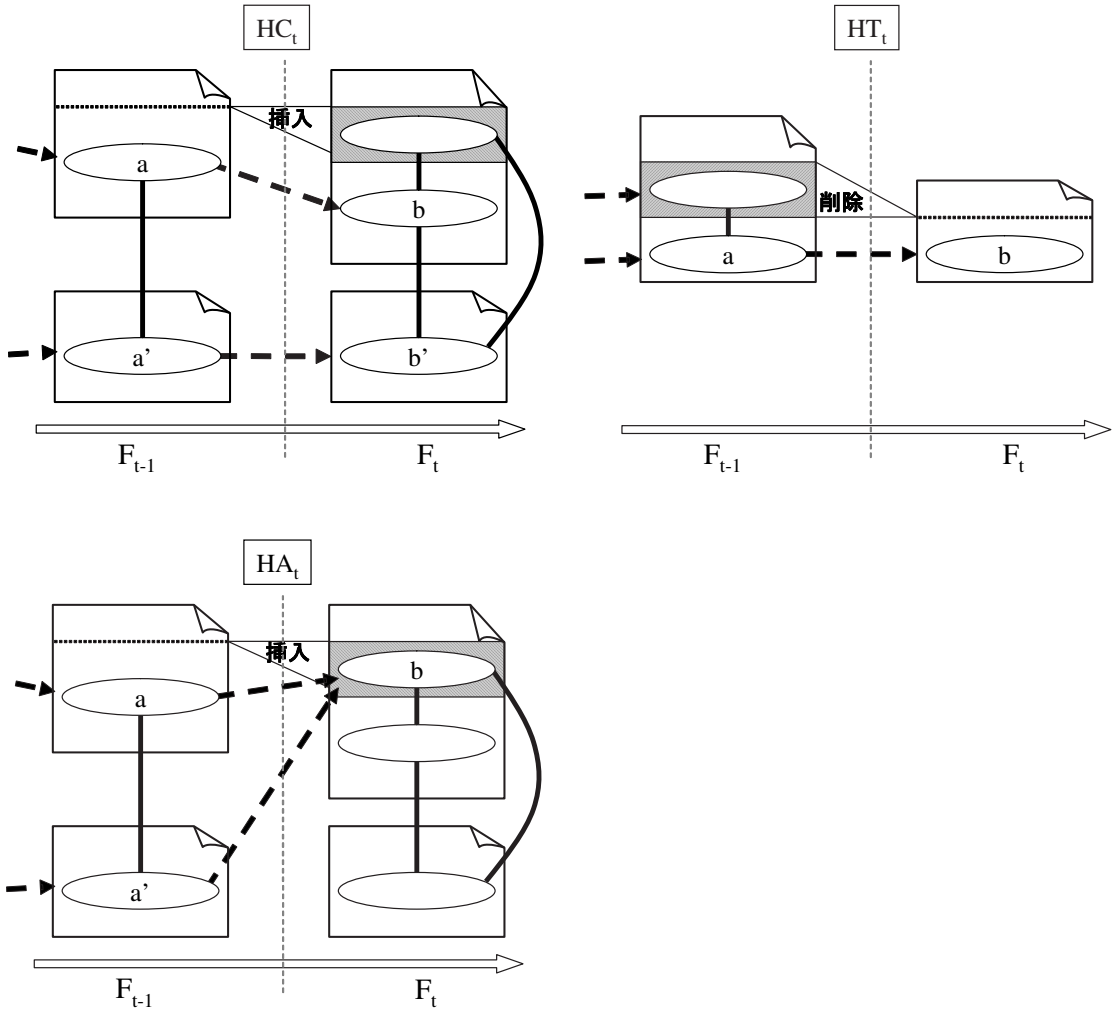


図 3.4: 3 種類の履歴関係

れは，既存クローンのうちどれか一つを原本と決めることは難しく，またそうすることの意義も薄いからである．

このように本手法では HA_t を HC_t と分けて定義している．HC_t では親子関係が明示的に一意に定まるのに対して，HA_t では親が属するクローンセット内のクローンすべてとクローン履歴関係が存在することになる．そのため，HA_t と HC_t を分けることでクローン履歴関係がより正確なものとなる．

HT_t は F_{t-1} においてクローンだったコード片のうち，F_t においてクローンペアをなすクローンが削除されたコード片 (図 3.4 中のクローン a_t) について，その対応関係を抽出するための定義である．このようなコード片は CCFinder によって検出されるクローンではないが，これも F_t におけるクローンとして扱う．このようなクローンについては，親子間においてテキスト類似度 (後述) が閾値以上ある場

合，これらの親子間に HT_t が成り立つものとする．

ただし，クローンペアに一度でもなった部分全てを追いつづけるのは非効率である．また，何らかの変更が加わった場合には，その時点で過去のコードとの関連は希薄になる．そこでテキストがほとんど変更されていない場合にのみクローン履歴対応関係があるものとする．

3.4 クローン履歴関係抽出手法

3.4.1 概要

以下にクローン履歴対応関係抽出手法の手順を示す．なお，クローン履歴解析を適用する期間の始まりを時刻 0，終わりを時刻 T で表し，C_t を F_t に関するクローンの集合，CT_t を HT_t を満たすコード片のうち F_t に属するコード片の集合とする．

0. $H_0 = \emptyset, CT_0 = \emptyset$.
1. $t = 0$ の F_0 を版管理システムから取得し，クローン解析手法を用いて C_0 を求める．
2. for $t = 1, 2, ..., T$

   (2-a) 版管理システムから F_t を取得し，クローン解析手法を用いて C_t を求める．

   (2-b) F_{t-1} に関する全てのクローンについて HC_t を求める．

   (2-c) $\{F_{t-1} \cup Change_{\Delta t}\}$ に対して CCFinder を適用して，HA_t を求める．

   (2-d) C_{t-1} の残りと CT_{t-1} について HT_t を求める．

このように，本手法では解析を行う期間 $[0 \cdots T]$ を Δt ごとに区切って考え，クローン履歴対応関係 $H_1, H_2, ..., H_T$ を順次，抽出する．

以下に各手順の詳細，および HC_t, HT_t の抽出時に利用する親子関係を定義するための写像について述べる．

3.4.2 各手順の詳細

本節では F_t の解析における各手順について，その詳細を述べる．なお，時刻 t における解析をはじめる時点で $t-1$ までの各情報 $(F_{t-1}, C_{t-1}, H_{t-1}, CT_{t-1})$ は既知であることに注意．

(2-a) 版管理システムから F_t を取得し，クローン解析手法を用いて C_t を求める

入力: (版管理システム)

出力: F_t, C_t

まず当該時刻のプロダクトを F_t を版管理システムから取得し，クローン解析を適用して C_t を得る．前述のとおり，本研究においてはクローン解析手法として CCFinder を用いる．

(2-b) F_{t-1} に関する全てのクローンについて HC_t を求める

入力: $F_{t-1}, F_t, C_{t-1}, C_t$

出力: HC_t

F_{t-1} に含まれる全てのクローンペアについて，それぞれのクローンの子が F_t に関するクローンペアとなっているかどうかを検証する．もしこの条件を満たしていれば，それぞれの親子ペアは HC_t に含まれるものとする．

(2-c) $\{F_{t-1} \cup Change_{\Delta t}\}$ に対して CCFinder を適用して，HA_t を求める

入力: $F_{t-1}, F_t, C_{t-1}, C_t$

出力: HA_t

次に F_{t-1} と $Change_{\Delta t}$ の間にクローン解析を適用する．そして $a \in F_{t-1}, b \in Change_{\Delta t}$ なるクローンペア a, b があれば，b が指す位置にあるクローン $b' \in C_t$ を考え (a, b') を HA_t に加える．

実際に適用する際には，編集された行とその前後 10 行を $Change_{\Delta t}$ とする．本研究では CCFinder が検出するクローンの最小トークン数を初期設定の 30 トークンで適用しているため，差分が 30 トークンよりも小さい場合にはクローンとして検出されなくなる．前後 10 行を含めることで，このような差分を含むクローンを適切に扱うことができる．

(2-d) C_{t-1} の残りと CT_{t-1} について HT_t を求める

入力: $F_{t-1}, F_t, C_{t-1}, HC_t, HA_t, CT_{t-1}$

出力: HT_t

最後に C_{t-1} に含まれるコード片のうち HC_t, HA_t 内のペアに含まれないコード片，および CT_{t-1} に含まれるのコード片 a について，その子 b とのテキスト類似度を計測して HT_t の抽出を行う．テキスト類似度は以下の式で定義する．

$$TextSim(a, b) = \frac{2|a \cap b|}{|a| + |b|}.$$

ただし $|a| = a$ の行数，$|a \cap b|$ は GNU diff によって同一と判定された行数とする．そしてテキスト類似度が 0.7 以上のコード片 b が存在したとき，$(a, b) \in HT_t$ とす

る．なお，コード片 a に対して子は複数ありうるため，HT_t の条件を満たすコード片も複数存在し得る．このときには類似度が最大のコード片のみを HT_t の対象とする．

このように HT_t の判定条件は，ほぼ同一のテキストかどうかの判定となっている．もしクローン自身も何らかの変更があった場合には，この部分は $Change_{\Delta t}$ に含まれるため HA_t の解析対象となる．従って HT_t の解析を行う際には，そのようなケースを考慮する必要はない．

3.4.3 コード片の写像

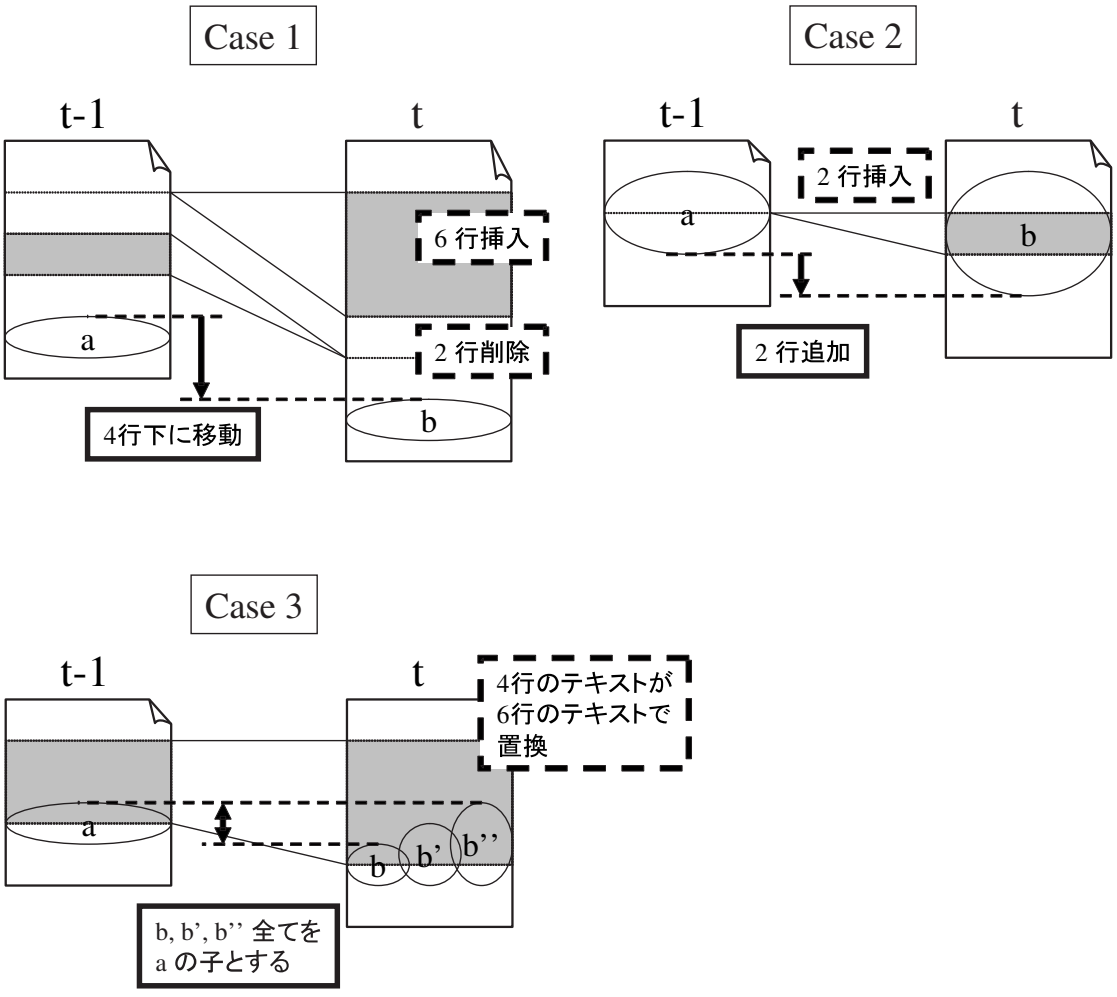


図 3.5: コード片の写像

本節ではコード片 $a \in F_{t-1}$ からコード片 $b \in F_t$ への写像について述べる．

まず時刻 $t-1$ と時刻 t において，コード片 a が含まれるファイルに変更がなかった場合，a の写像先 b の開始行，終了行は a と全く同じとする．

次に，コード片 a が含まれるファイルに何らかの編集操作が行われていた場合を考える．もし a よりも前の部分に変更箇所があれば，その内容に応じて写像先 b の開始行，終了行を調整する．図 3.5 の Case 1 はコード片 a の前で編集操作が行われた場合の例である．このとき，a の前で行われた編集操作の全てを勘案して a の写像先 b の開始行，終了行は a のそれのそれぞれ 4 行後ろとする．また a に含まれる部分に変更が加えられていた場合には，その内容に応じて b の終了行を調整する．図 3.5 の Case 2 では a 内に 2 行新しい行が追加されているため，b の終了行は a のそれに対して 2 行追加した値とする．

最後に，クローン片の端の部分で書きかえ操作が発生した場合について述べる．図 3.5 の Case 3 ではコード片 a の開始行を跨がる形で書きかえされている．このような場合,「コード片 a の上に 2 行挿入があった」という解釈と,「コード片 a に 2 行の挿入がされた」という解釈，およびその中間 (1 行がコード片の上に，1 行はコード片 a 中に挿入された) が成りたつ．そこで，これら全てのケースをコード片 a の子とする．このように複数の子が生成されるケースは以下の 3 つである．

- 開始行，終了行が編集されている箇所を含む (図 3.5 Case 3)
- 開始行の一行前に挿入
- 終了行に挿入

本研究においては $F_{t-1} \rightarrow F_t$ への写像のみを扱うため，削除操作ではこのような複数候補を考える必要はない．

以上述べた 3 つの編集操作による影響をすべて足しあわせて写像先を決定する．

3.5 実験

3.4 で述べた手法を用いることで図 3.1 におけるクローンセット B, C 間のような分岐クローンセットを抽出することができる．本節では，PostgreSQL を対象として本手法を適用し，実際に分岐クローンセットをどの程度抽出できているかの検証を行う．また，クローン履歴分析の過程で得られるソースコード全体に含まれるクローン量の変遷グラフから得られる知見についても取りあげる．

PostgreSQL を対象とした理由は，第一に提案手法で用いている CVS リポジトリが公開されていること，第二に PostgreSQL が非常に広く活用されており，開発も活発に行われているからである．

3.5.1 分岐クローンセットの抽出

実験方法

PostgreSQL リポジトリのなかから，ソースコードか格納されている src ディレクトリ以下の全てのファイルを対象とした．これらのファイルを 2005/01/01 〜 2005/6/30 までの 6 ヶ月間について Δt を 1 週間としてクローン履歴関係の抽出を行った．

そして各バージョンに含まれるクローン c_t，および c_t とクローン履歴対応関係にある c_{t-1} について，c_t が属するクローンセットに含まれるクローン数が，c_{t-1} のそれに対して減少している箇所を，”クローン減少箇所” として抽出した．

次に，それぞれのクローン減少箇所において，そこに含まれるクローンセットが分岐クローンセットかどうかの判定を人手により行い，もし分岐クローンセットであればクローン履歴関係を正しく抽出できているかどうか検証した．

このように本実験ではクローンセット内のクローン数が減少が確認できたクローンセットのみに着目した．これは，実際に開発されているソフトウェアには数百から数万もの大量のコードクローンが存在し，さらに CCFinder では検出できない潜在的に存在するコードクローンもかなりの数に上るため，全てのクローンセットを検証することは現実的ではないからである．

実験結果

Branched clone set	4
Branched clone set (miss)	0
Deleted clone set	5
Useless clone set	15

表 3.1: クローン減少箇所の調査結果

実験手法を適用した結果，クローン減少箇所は計 24 箇所あった (表 3.1)．そのうち 4 箇所がクローンが分岐クローンセット (Branched clone set) であり，4 箇所ともクローン履歴関係も適切に抽出できていることがわかった．5 箇所がクローンセット内のクローンのいくつかが削除されたもの (Deleted clone set)，残り 15 箇所はそもそもクローンとして有用でないものであった (Useless clone set).

有用でないクローンとは，printf 文が連続している部分や数十，数百の定数定義が続いている部分などのように，単純な構文が 10 行以上連続している部分を指す．CCFinder は構文木として同形であればクローンとして検出を行うため，このよう

な単純な構文が繰りかえされている箇所もクローンとして検出される．しかし，このような部分はそもそもクローンとして検出する必要がなく，実際にユーザにとっても中身を検討する価値がない．したがって，本実験においてはそのような箇所を評価から除外している．

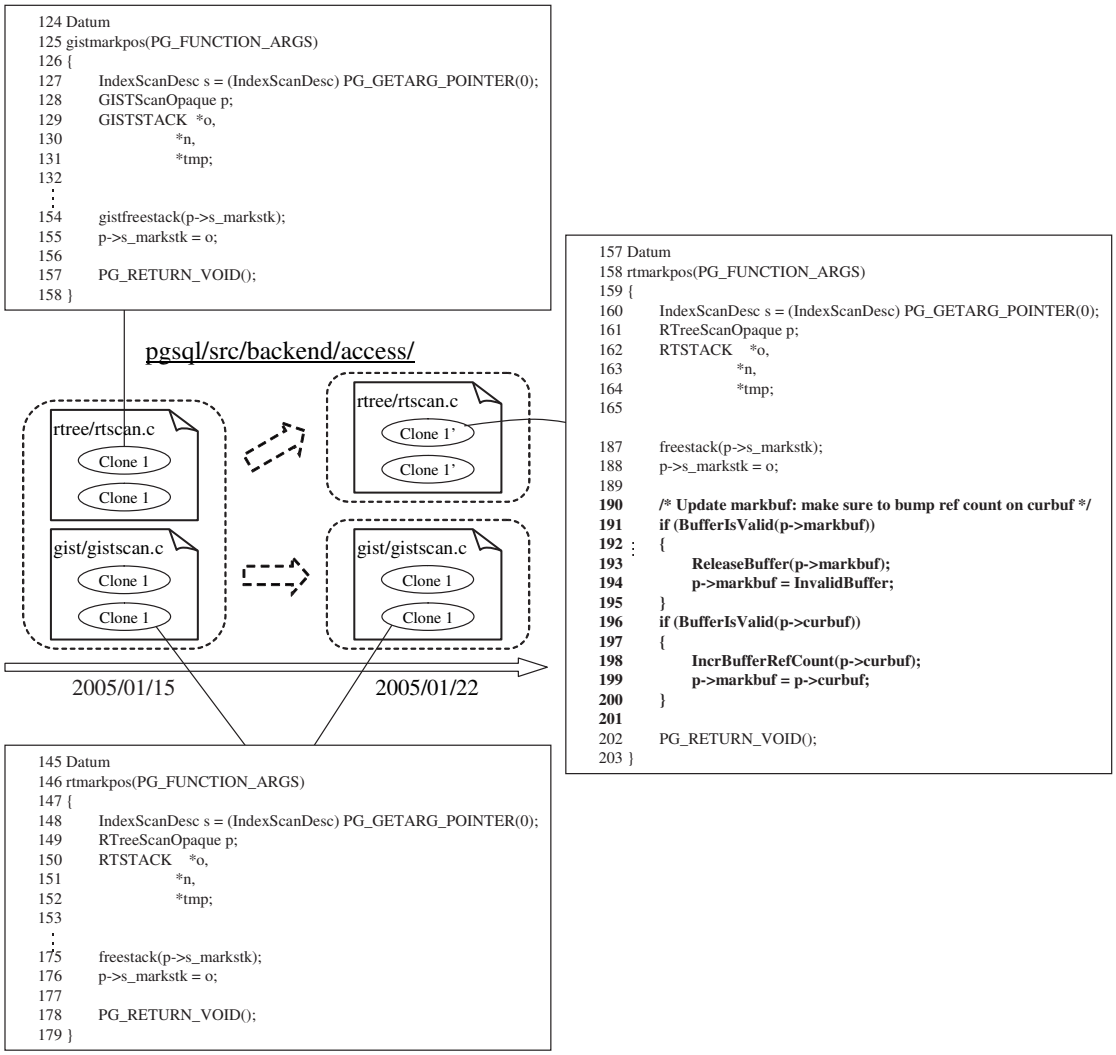


図 3.6: 過去にクローンペアであったコード片の例

図 3.6 に分岐クローンセットとなったクローン減少箇所の例を示す．2005 年 1 月 15 日の時点では 4 つのコード片がクローンであった．一週間後，そのうちの 2 つに編集が加えられ，それぞれ独立したクローンセット Clone 1 と Clone 1' になった．図 3.6 右側は新しい Clone 1' の抜粋である．このように Clone1' は太字で示した処理が追加されただけで，本質的には Clone1 同様のことを行っており，これらのコード片の間には依然として強い類似性がある．本手法により過去の履歴を

辿ることで，これらのコード片の関係を抽出することができる．

3.5.2 クローン量変遷グラフ

実験方法

次に PostgreSQL の開発工程においてクローンの量がどのように変遷したかをクローンの行数，およびクローンの割合という二つの観点から分析した．分析は 1998 年 7 月から 2005 年 6 月までの 7 年分のデータに対して Δt を一ヶ月として実施した．

分析対象は PostgreSQL 全体と，src/backend 以下のディレクトリを選んだ．src/backend 以下を選んだのは，PostgreSQL の主要なソースコードが src/backend 以下に収められているからである．実際に，PostgreSQL ではプロダクトの大部分を src ディレクトリが占めており，その中でも backend ディレクトリ以下は src ディレクトリの約 7 割のソースコードが格納されている．

実験結果

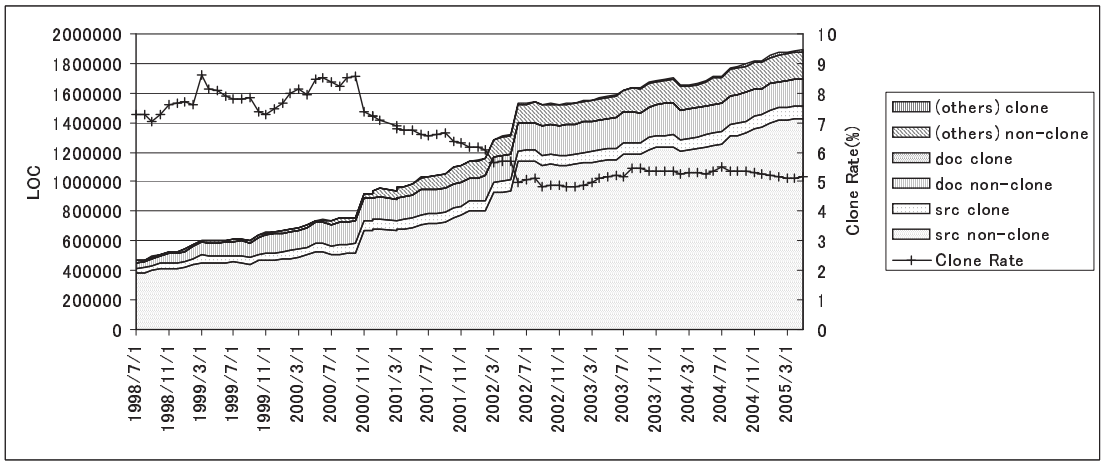


図 3.7: PostgreSQL の各サブディレクトリの LOC およびクローン無しの LOC の推移

図 3.7 に PostgreSQL 全体の，図 3.8 に src/backend ディレクトリのクローン行数，非クローン行数を示す．またクローン含有率については，PostgreSQL 全体のクローン含有率を図 3.7 中の実線として，src/backend 以下の各サブディレクトリのそれを図 3.9 に示す．

図 3.7 を見ると，クローンの比率は初期には少しずつ増えてるものの，開発が進むにつれてクローンの占める割合が徐々に低下していることがわかる．これは既存のコードが正しく再利用されていること，すなわち開発されたソースコードの品質が高いことを示唆している．

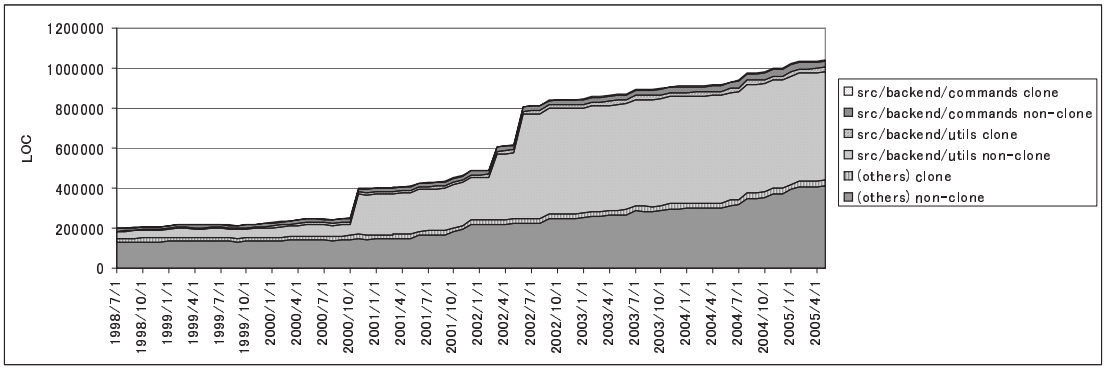


図 3.8: src/backend の LOC およびクローン無しの LOC の推移

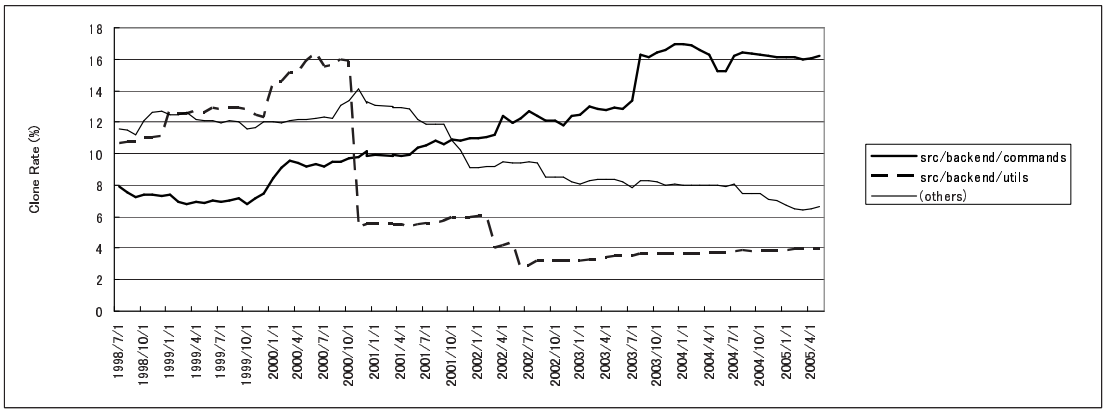


図 3.9: src/backend 以下のサブディレクトリのクローンの割合

次に，src/backend ディレクトリについての分析結果を示す．図 3.8 に行数を，図 3.9 にクローンが全体に占める割合の変化を示す．図 3.8，3.9 から 2000 年 10 月の時点で utils 以下のファイルに大量のコードが追加されていること，追加されたコードの中にはクローンが含まれていないことなどが読みとれる．実際に差分を確認したところ，この間に文字コード変換機能が追加されており，utils 以下には文字コード間の対応表が追加されていたことが確認できた．

逆に SQL 命令を実際に処理する部分である src/backend/commands ディレクトリでは徐々にクローンの割合が増加している．特に上昇の著しい 2003 年 7 月と 2003

年 8 月の間の変更を調べたところ，エラーを処理するための変更として同様のコードが追加されており，それらがクローンとして判定されていることがわかった．

このように，クローン量の変化を調査することで開発工程において異常にクローンが増えていないかを監視したり，モジュールごとの傾向を調査したりすることができる．PostgreSQL を対象とした調査では開発が適正に進んでいるという推測ができるが，たとえばある時点において急激にクローンが上昇していたりしていれば，これを調査することができる．

3.6 関連研究

クローンの履歴を調査する研究としては，Kim らの研究 [37] が挙げられる．Kim らは我々の手法と同様に CCFinder を用いてクローンの履歴抽出を試みており，クローンセットの生存期間に着目して生存期間の分布がどのようになっているか，生存期間の違いによってクローンにどのような特徴があるかを調査している．Kim らの手法では，クローン間の履歴を抽出するために F_{t-1} と F_t 全体を CCFinder を用いて解析している．そのためソフトウェアが大規模になればなるほど計算時間，および必要なメモリ空間が非常に多くなる．特に時刻 t で編集が行われていない部分はすべてクローンとして検出されるため Δt を小さくとればとるほど効率の悪化が著しい．そのため大規模なソフトウェアに適用するためには，小規模なクローンを無視し，Δt も大きな値にすることで計算コストを抑える必要がある．我々の提案手法では F_{t-1} と $Change_{\Delta t}$ についてのみ CCFinder を適用しているため，CCFinder が適応できる規模であれば解析が可能である．

任意のコード片ではなく，関数を対象とする履歴追跡手法もいくつか提案されている [26, 27]．Godfrey ら [26] は関数のペアを対象とした 5 つの評価基準を定義している．これらは関数名の類似度，関数宣言部の類似度，LOC 等のメトリクスの類似度，呼び出し元や呼び出し先の類似度，および上記 4 つをユーザが組みあわせたものから構成され，ユーザが指定した計算尺度に基づいてバージョン間の関数同士の対応関係を求めている．また Antoniol ら [3, 4] は，クラス単位での履歴追跡手法を提案している．

このほかの関連研究として版管理システムを利用して何らかの有益な情報を引きだそうとする研究が挙げられる．これまでに，版管理システムに存在する開発履歴から関連性の高い関数を提示する研究 [65, 18] や，バグ修正支援 [62]，行数の変化や開発者ごとの編集を行った行数などを図示する試み [21, 17] が行われている．

3.7 結論と課題

本論文では，クローン履歴解析手法を提案した．また PostgreSQL を対象に適用実験を行い，クローン履歴関係を用いて，かつてクローンだったコード片が正しく抽出できること，抽出したコード片は派生元のコード片と強い関連を持っていることを確認した．また，クローン量の変遷といった有用なデータが得られることを示した．

今後の課題としては，まず，より目的に沿った形でのクローン履歴情報閲覧システムの構築が挙げられる．クローンの履歴は開発プロセス全体の把握やクローン発生源の特定，クローンの管理等さまざまな応用が考えられるが，抽出結果を活用するためには分析した結果を目的に沿う形で表示する必要がある．

また分析手法そのものについても評価，改善が必要である．クローンの履歴となりうるもののうち本手法で分析できる範囲，できない範囲を厳密に調査する必要がある．また既存の類似手法との結果比較も重要であると考える．

現時点で判明している取得できないクローン履歴としては，一度削除されたクローンが，その後の変更により復活した場合がある．前述したとおり全てのバージョン間においてクローン履歴分析をするのは現実的ではないため，たとえば削除されたクローンについても逐次記録していき，過去の削除されたクローンも差分解析の対象として加えることが考えられる．しかし，この手法は削除されたクローン数によっては膨大な計算量を要するため，既存の開発パターンを調査して削除されたクローン数の推移傾向を調査する必要があるであろう．

第4章　むすび

4.1　まとめ

本研究ではソフトウェアの類似性に着目し，そこに存在する問題点 2 つの解決を試みた．

MUDABlue システムでは，既存の手法の多くがソフトウェアコンポーネントなどのより細粒度の単位についてのみ着目しているのに対して，ソフトウェアそのものを単位とした類似度を定義し，ソフトウェアを自動的に分類する手法を提案し，それをシステムとして実装した．MUDABlue システムではソースコードを文書と見做して潜在的意味解析手法 LSA を適用し，カテゴリの自動抽出を行う．本手法はソースコードのみから分類を行う．従来の手法は付属する文書を必要としていたのと比較して本手法の適用範囲はより広範囲といえる．また分類先であるカテゴリも自動的に抽出を行う．カテゴリの定義はソフトウェアアーキテクチャや既存のライブラリについて広汎な知識を必要とするが，MUDABlue ではそのような知識を一切依存せずに分類を行うことを可能にした．実験の結果，従来手法と比較して適用範囲が大幅に広がっているにもかかわらず，従来手法と同程度の精度を実現していることを確認した．また分類結果を効率的に閲覧するためのユーザインタフェースを作成した．

次にクローン履歴分析では，最新状態において類似しているかどうかだけでなく，過去の時点において類似していたことがあるというのも有用な情報であることに着目し，クローン履歴分析手法を提案した．本手法によって取得できるクローン履歴によって過去に類似関係にあったコード片が容易にとりだせること，そうして取りだしたコード片の間には深い関係があることを実験により確認した．またクローン分析を逐次的に適用することで得られるクローン量の変化をグラフ化することによって開発過程において注意すべき変化を確認することができることを示した．

4.2 今後の研究方針

今後の研究方針としては，より広汎かつ有用な類似度の抽出を定義・実装することを考えている．

まずソフトウェア間の類似度についてであるが，現在の MUDABlue の実装では最新版同士の比較を行っている．そのため，各ソフトウェアが開発の初期段階であるのか，開発途上なのか，すでに主な機能の実装が終了した保守段階であるのかは考慮されておらず，全く異なる開発段階にあるソフトウェアを同じ基準で比較することになる．MUDABlue で採用している手法はソースコードに依存しているため，このように各段階においてソースコードの量が異なるソフトウェアを比較した場合には適正な結果とならない可能性がある．そこで，各ソフトウェアの同一の段階同士を比較することによって，より適正な分類を行うことが期待できる．

つぎにクローンについてであるが，クローン検出手法によって検出されるクローンが大量に存在する場合，それらクローンの分析が困難になるという問題が指摘されている．本論文で提案した手法もこれまでの手法では抽出できなかった類似性の抽出を行うためさらにこの問題を深刻化させてしまう．そこで，検出されたクローンを出現場所や行数などの傾向からグループ分けを行い優先的に注目すべきクローンを提示することでクローン分析作業を支援することができるのではないかと考えている．

参考文献

[1] J. Allan, A. V. Leouski, and R. C. Swan. Interactive cluster visualization for information retrieval. Technical Report IR-116, Center for Intelligent Information Retrieval, University of Massachusetts, Amherst, 1997.

[2] N. Anquetil and T. Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *Proc. Int. Conf. on Software Engineering,(ICSE'98)*, pp. 84–93, Apr 1998.

[3] G. Antoniol, G. Casazza, M. Penta, and E. Merlo. Modeling clones evolution through time series. In *Proc. IEEE Intl. Conf. on Software Maintenance 2001(ICSM 2001)*, pp. 273–280, Fiorence, Italy, Nov 2001.

[4] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proc. 7th Int. Workshop on Principles of Software Evolution (IWPSE'04)*, pp. 31–40, Kyoto, Japan, Sep 2004.

[5] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[6] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc.2nd Working Conf. on Reverse Engineering (WCRE95)*, pp. 86–95, Los Alamitos, CA, Jul 1995.

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

[8] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. 6th IEEE Int. Symposium on Software Metrics (METRICS99)*, pp. 292–303, Nov, Boca Raton, Florida, USA 1999.

[9] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *Proc. 6th Int. Working*

Conf. on Reverse Engineering (WCRE'99), pp. 326–336, Oct, Atlanta, Georgia, USA 1999.

[10] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. 7th Working Conf. on Reverse Engineering (WCRE 2000)*, pp. 98–107, Brisbane, Queensland, Australia, Nov 2000.

[11] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. IEEE Int. Conf. on Software Maintenance 1998(ICSM'98)*, pp. 368–377, Bethesda, Maryland, Nov 1998.

[12] M. W. Berry, T. Do, G. W. O'Brien, V. Krishna, and S. Varadhan. SVDPACKC (Version 1.0) User's Guide. Technical Report CS-93-194, University of Tennessee, Knoxville, TN, April 1993.

[13] Y. Brun. Software fault identification via dynamic analysis and machine learning. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, Aug 2003.

[14] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proc. 2nd IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pp. 36–43, Montreal, Canada, Oct 2002.

[15] A. Chan and T. Spracklen. Feature indicators: A self-organising map approach to legacy code. In *Proc. Int. Conf. on Artificial Intelligence (IC-AI'2000)*, pp. 1449–1454, Las Vegas, Nevada, USA, June 2000.

[16] K. Chen and V. Rajlich. Case study of feature location using dependency graph. In *Proc. 8th Int. Workshop on Program Comprehension (IWPC'00)*, pp. 231–239, Limerick, Ireland, June 2000.

[17] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proc. the 2003 ACM symposium on Software visualization (SOFTVIS 2003)*, pp. 77–86, San Diego, California, USA, Jun 2003.

[18] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Software Engineering*, 31(6):446–465, Jun 2005.

[19] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *J. Am. Soc. Inf. Sci.*, 41(6):391–407, 1990.

[20] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. IEEE Int. Conf. on Software Maintenance 1999 (ICSM'99)*, pp. 109–118, Oxford, UK, Aug 1999.

[21] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE Trans. Software Engineering*, 28(4):396–412, Apr 2002.

[22] C. Fluit, M. Sabou, and F. van Harmelen. Supporting user tasks through visualisation of light-weight ontologies. In S. Staab and R. Studer eds., *Handbook on Ontologies in Information Systems*, pp. 415–434. Springer-Verlag, 2003.

[23] W. B. Frakes and T. Pole. An empirical study of representation methods for reusable software components. *IEEE Trans. Software Engineering*, 20(8):617–630, 1994.

[24] freshmeat.net. `http://freshmeat.net/`.

[25] GForge. `http://www.gforge.org/`.

[26] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Software Engineering*, 31(2):166–181, Feb 2005.

[27] N. Gold and A. Mohan. A framework for understanding conceptual changes in evolving source code. In *ICSM '03*, pp. 22–26, Amsterdam, 2003.

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

[29] D. Harman. An experimental study of factors important in document ranking. In *Proc. ACM Conf. on Research and development in information retrieval*, pp. 186–193, Pisa, Italy, September 1986.

[30] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[31] 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 年 9 月.

[32] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proc. the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pp. 171–183, Toronto, Ontario, Canada, Oct 1993.

[33] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Engineering*, 28(7):654–670, 2002.

[34] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Automatic categorization algorithm for evolvable software archive. In *Proc. 2003 Int. Workshop on Principles of Software Evolution(IWPSE 2003)*, pp. 195–200, Helsinki, Finland, Sep 2003.

[35] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. In *Proc. 11th Asia-Pacific Software Engineering Conf.(APSEC2004)*, pp. 184–193, Busan, Korea, Nov. 2004.

[36] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[37] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR 2005*, pp. 17–21, Saint Louis, Missouri, May 2005.

[38] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. 8th Int. Symposium on Static Analysis*, pp. 40–56, Paris, France, Jul 2001.

[39] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. 8th Working Conf. on Reverse Engineering (WCRE2001)*, pp. 562–584, Stuttgart, Germany, Oct 2001.

[40] T. K. Landauer and S. T. Dumais. A solution to plato's problem: The latent semantic analysis theory of the acquisition, induction, and representation of knowledge. *Psychological Review*, 104(2):211–240, 1997.

[41] T. K. Landauer, P. W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25:259–284, 1998.

[42] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proc. the Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, pp. 289–302, 2004.

[43] G. A. D. Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. D. Carlini. Comprehending web applications by a clustering based approach. In *Proc. 10th Int. Workshop on Program Comprehension(IWPC'02)*, pp. 261–270, Paris, France, June 2002.

[44] G. Lucca, M. D. Penta, and S. Gradara. An approach to classify software maintenance requests. In *Proc. Int. Conf. on Software Maintenance (ICSM'02)*, pp. 93–102, Montreal, Quebec, Canada, Oct 2002.

[45] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Software Engineering*, 17(8):800–813, 1991.

[46] J. I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proc. 12th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI'00)*, pp. 46–53, Nov. 2000.

[47] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. 25th Int. Conf. on Software Engineering(ICSE2003)*, pp. 125–135, Portland, OR, May 2003.

[48] J. Mayland, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. IEEE Int. Conf. on Software Maintenance (ICSM'96)*, pp. 244–253, Monterey, CA, USA, Nov 1996.

[49] S. McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004.

[50] E. Merlo, G. Antoniol, M. D. Penta, and V. F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analyses. In *Proc. 20th IEEE Int. Conf. on Software Maintenance (ICSM'04)*, pp. 412–416, Chicago, Illinois, USA, Sep 2004.

[51] 門田, 佐藤, 神谷, 松本. コードクローンに基づくレガシーソフトウェアの品質の分析. 情報処理学会論文誌, 44(8):2178–2188, 2003 年 8 月.

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

[53] 酒井, 山口, 川合. 図形オブジェクトの遠隔度に基づく階層集合の可視化モデル. 情処学論, 40(9):3455–3470, 1999 年 9 月.

[54] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. 2003 ACM SIGMOD Int. Conf. on Management of Data*, pp. 76–85, San Diego, CA, Jun 2003.

[55] SourceForge.net. `http://sourceforge.net/`.

[56] SourceShare. `http://www.zeesource.net/`.

[57] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[58] A. Spoerri. Infocrystal: a visual tool for information retrieval. In *Proc. 2nd Int. Conf. on Information and Knowledge Management*, pp. 11–20, Washington, D.C., United States, Nov 1993.

[59] Tigris.org. `http://www.tigris.org/`.

[60] TouchGraph. `http://www.touchgraph.com/`.

[61] S. Ugurel, R. Krovetz, C. L. Giles, D. M. Pennock, E. J. Glover, and H. Zha. What's the code? automatic classification of source code archives. In *Proc. 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pp. 632–638, Edmonton, Alberta, Canada, Jul 2002.

[62] C. C. Willams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Software Engineering*, 31(6):466–480, Jun 2005.

[63] 山本, 松下, 神谷, 井上. ソフトウェアシステムの類似度とその計測ツール smmt. 電子情報通信学会論文誌 D-I, Vol. J85 － D － I(No.6):503–511, 2002 年 6 月.

[64] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proc. 24th Int. Conf. on Software Engineering (ICSE 2002)*, pp. 513–523, Orlando, Florida, USA, May 2002.

[65] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Engineering*, 31(6):429–445, Jun 2005.