修士学位論文

題目

コードクローンの集約によるリファクタリング支援システムの提案と実装

指導教官
井上克郎 教授

報告者
肥後芳樹

平成 16 年 2 月 12 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 15 年度 修士学位論文

コードクローンの集約によるリファクタリング支援システムの提案と実装

肥後芳樹

内容梗概

近年，ソフトウェアの大規模化・複雑化に伴い，保守作業に要するコストは増大している．ソフトウェアの保守を困難にしている要因の 1 つとしてコードクローンがあげられる．コードクローンとはソースコード中に存在する同一，または類似したコード片のことである．例えば，あるコード片にバグが含まれていた場合，そのコード片のコードクローン全てについて修正の是非を考慮する必要がある．コードクローンを対象とする保守支援としては，ソフトウェア内に存在するコードクローンを把握，管理する方法と，ソフトウェアからコードクローンを取り除く（リファクタリング）方法の 2 つがあげられる．前者については我々はこれまでに，コードクローン分析環境 Gemini を開発し，さまざまな事例に対して適用してきた．一方，後者に対してはこれまでにいくつか提案がされているが，時間的コストの非常に高いものであったりと，実際に社会で用いられているソフトウェアに対して適用可能な手法は存在しない．本研究では，高速なコードクローン検出ツール CCFinder を利用することで，大規模ソフトウェアから実用的な時間でリファクタリングに適したコードクローンの抽出を試みる．また，抽出したコードクローンの特徴をメトリクスとして数値化することにより，その集約方法の補助も行なう．提案手法により，ユーザは実際のソースコード修正作業に集中でき，効率的なソフトウェア保守作業を行なうことができると期待される．また提案手法を実装したツールを作成し，適用実験を行なうことで，本手法の有用性を確認した．

主な用語

コードクローン
ソフトウェア保守
リファクタリング
オブジェクト指向

目 次

1　まえがき

近年，ソフトウェアの大規模化・複雑化に伴い，ソフトウェアの保守作業がますます困難になってきている．ソフトウェア保守では，フィールドバグの修正，環境変化に対する機能追加・変更， 将来トラブルにつながりそうな箇所に対する対応等の作業が実施される [35]．しかし，保守対象のソフトウェアがうまく設計されていない，度重なる変更により構造がわかりにくくなってしまう，変更履歴のドキュメントが存在しない等の問題により，ソフトウェア保守コストは増大してきている．実際に，多くのソフトウェア会社が既存システムの保守に非常に多くの人的，時間的コストをかけていると報告されている [39]．

コードクローンはソフトウェア保守を困難にしている 1 つの要因といわれている [15]．コードクローンとはソースコード中に存在する同一，または類似したコード片のことである．コードクローンが生成される原因はさまざまな理由が考えられるが，その最も大きな原因の 1 つとしてコピーアンドペーストによる修正，拡張作業があげられる．コード片にバグが含まれていた場合，そのコード片のコードクローンとなっている部分全てに対して修正の是非を考慮する必要がある．このような作業は，特に大規模ソフトウェアでは非常に手間のかかる作業である．従ってコードクローン検出の効率化はソフトウェア保守工程の改善において有効である．これまでにコードクローンを自動的に発見するためのさまざまな手法が提案されている [4][5][6][10][13][27][28][30][31][32][33][34]．

その手法の 1 つとして，我々はコードクローン検出ツール CCFinder[28] と分析環境 Gemini[36][37] を開発してきている．ユーザは Gemini を用いることによりコードクローンの解析，ソースコードの修正を容易に行なうことができる [38]．Gemini は主に，クローン散布図とメトリクスグラフをユーザインターフェースとして提供する．クローン散布図はソースコード中のコードクローンの分布状態を俯瞰的に表示する．またメトリクスグラフは各々のコードクローンについての定量的な情報を提供し，その値を用いることによって保守を阻害するコードクローンの選択をすることが可能である．選択されたコードクローンのソースコードは容易に閲覧できる．ユーザはこれらの機能を用いることによってソフトウェアの保守作業を改善することができると期待できる．

このように多くのコードクローンを把握，管理する手法が提案されている一方で，ソフトウェアからコードクローンを取り除く研究はそれほど活発に行なわれてはいない．これまでにいくつかは提案がされているのであるが，それらの手法は時間的コストが非常に高いものであったりと，実際のソフトウェア開発，保守現場での適用は困難である．

本研究では，中規模〜大規模ソフトウェアに対して実用的な時間で適用可能なリファクタリングを対象としたコードクローンの検出，集約手法を提案する．そして，提案手法に基づくリファクタリング支援ツール Cancer の試作を行なう．高速なコードクローン検出ツール

CCFinder を用いることにより，大規模なソフトウェアからでも実用的な時間でリファクタリングの適用単位となるコードクローンの検出を行なう手法を提案する．また，検出したコードクローンの特徴をメトリクスとして数値化することにより，そのコードクローンの集約方法の補助を試みる．

提案手法を用いることより，ユーザはソフトウェアのどの部分がリファクタリングできるのか，またその部分がどのようにリファクタリングできるのかの補助を受けることができる．これによりユーザは実際のソースコード修正作業に集中することができ，効率的な保守作業を行なうことができると期待できる．また，Cancer を用いて中規模ソフトウェアに対して行なった実験結果では，リファクタリングすべきコードクローンとして検出された 15 個のクローンクラスの内，12 個のクローンクラスについて実際にコードクローンを集約することに成功した．

以降，2 章では，コードクローン関する諸定義，コードクローン検出ツール CCFinder，コードクローン分析環境 Gemini，コードクローンに関するメトリクス等について説明する．3 章では，リファクタリングを目的としたコードクローン解析手法について詳しく説明する．4 章では，3 章で提案した解析手法を実装したツールに付いて述べ，5 では作成したツールを用いて行なった適用実験について説明する．最後に 6 章で，まとめと今後の課題を述べる．

2　ソフトウェア保守とコードクローン

2.1　ソフトウェア保守

2.1.1　ソフトウェア保守と保守における問題点

ソフトウェア保守とは，“納入後，ソフトウェア・プロダクト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” であると定義されている [22]．また，目的毎に次の 4 つのカテゴリに分けられている．

修正のための保守 (Correction) 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，

適応のための保守 (Adaptation) 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，

完全化のための保守 (Perfection) 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，

予防のための保守 (Prevention) 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．

ソフトウェア保守は，ソフトウェアライフサイクルコストの大きな部分を占めており [14]，ライフサイクルを考えるとき，その費用と労力からみて保守は非常に大切な工程である．しかし，ソフトウェア保守における課題は多くあり，Dorfman および Thayer[12] は，保守に関しては，投資効果が明らかにならないので，常に資源を獲得するための戦いが起きると述べている．資源を巡って争うということが常に存在し，次のソフトウェア・プロダクトに対するコードを作成しながら，将来の納入計画を決め，現在のソフトウェア・プロダクトに対して緊急的なパッチを施すということは難題である．また，ソフトウェアを開発したチームは，ソフトウェアが運用に入ると必ずしも保守を担当するとは限らない．自分の開発したものではない，大規模なソースコードに潜む欠陥を発見しなければならないということは，保守者にとっては非常に難題である．

多くの場合，独立したチームが，ソフトウェアが適切に運用されユーザの変化するニーズを満たすように進化させられていることを確実にするために雇用されているが，保守のアウトソーシング (社外調達) も，主要な産業になりつつある．大企業は，非公開としたいビジネス中核となるソフトウェアを除いては，ソフトウェア保守を含めて，運用をアウトソーシ

ングする．このような背景のために，開発者は通常コードを説明しなければならない場には居合わせないことが多く，変更も文書化されていないことも多い．したがって，保守者は，ソフトウェアに関して制限された理解しか得ることができず，ソースコードを自分で読まねばならない．文献 [35] では保守作業のおおよそ 40%から 60%は，改変すべきソフトウェアの理解に費やされていると指摘されている．したがって，保守作業の効率を高めること，さらにプログラムの理解容易性を高めるということは，ソフトウェア工学における重要な課題の一つとなっている．

2.1.2 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 [15].

ソフトウェアは，開発者によるコードの変更が短期的な視野にたったものであったり，設計の全体的な理解をせずに行われたりすると，コードは構造を崩し，コードを読んで構造を把握することも難しくなる．また，設計が把握しづらくなるにつれ，それを維持するのが困難になり品質の劣化を招くこととなる．それに対して，リファクタリングを行うことにより，ソフトウェア設計品質を向上させ，ソフトウェアは理解しやすくなる．同時に，コードが理解できるようになると，プログラムが何を行っているのかが明確になりバグを見つけやすくなる．特に，ロジックが重複しているプログラムは変更が難しい (Programs that have duplicate logic are hard to modify) と言われている [15]．重複がある限り，一方への修正が他方に反映されないことになるというリスクに直面しており，将来のバグの一因となる可能性があるからである．そういった重複したコードに対するリファクタリング手法としては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).

メソッドの抽出

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 参照).

メソッドの引き上げ

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 2 参照)．重複したコードが兄弟クラスに存在した場合には，メソッドの抽出を行ってから，メソッドの引き上げを行えばよい.

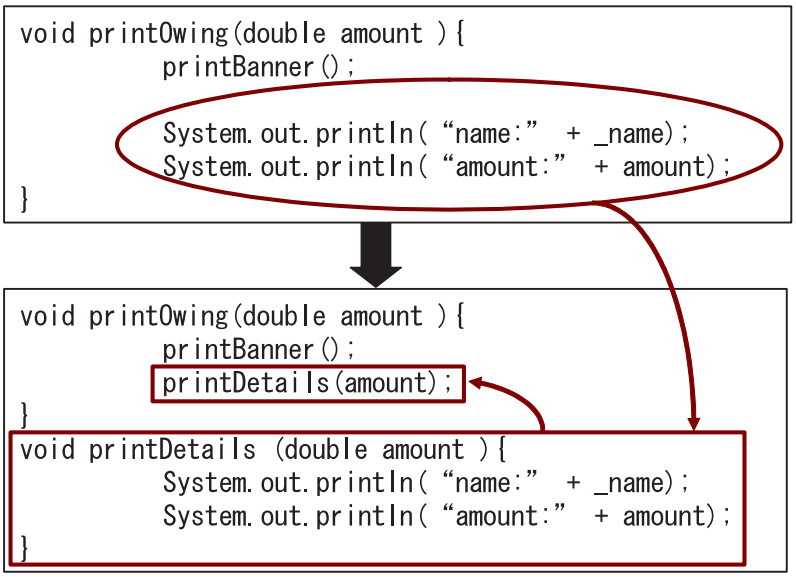


図 1: リファクタリング（メソッドの抽出）

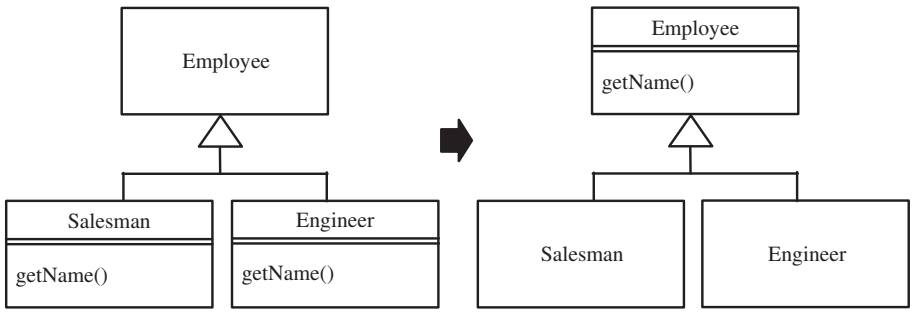


図 2: リファクタリング（メソッドの引き上げ）

クラスの抽出

2 つのクラスでなされるべき作業を 1 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．

スーパークラスの抽出

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，共通の特性を移動する．クラスの抽出との違いは，継承するか委譲するかの違いである (図 3

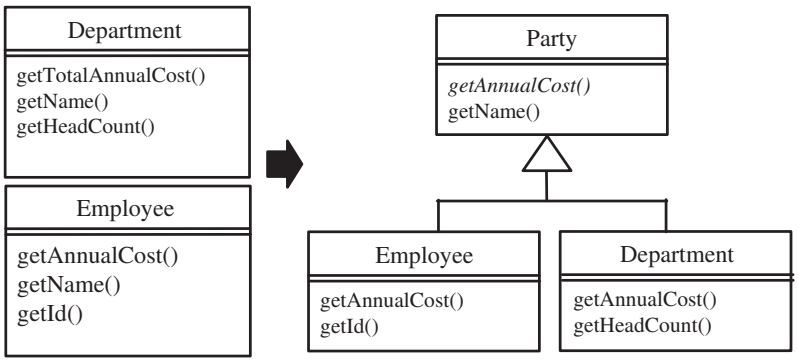


図 3: リファクタリング (スーパークラスの抽出)

参照).

テンプレートメソッド (TemplateMethod) の形成

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 4 参照).

重複した条件記述のコード片の統合

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 5 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.

ポリモーフィズムを用いた switch 文の削除

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メソッドの抽出等で対処することができる.

保守プロセスにおいてリファクタリングは，特に機能追加や，バグ修正，コードレビューの際に行うのがよいとされている．いずれもコードの理解が必要な作業であり，リファクタリングを行うことで，コードの理解が深まり，バグが混入しにくくなり，バグを発見しやすくなる．しかし，リファクタリングとは，あくまでもソフトウェアを理解しやすく，変更を容易にするために行うことであり，機能追加とは区別されなければならない.

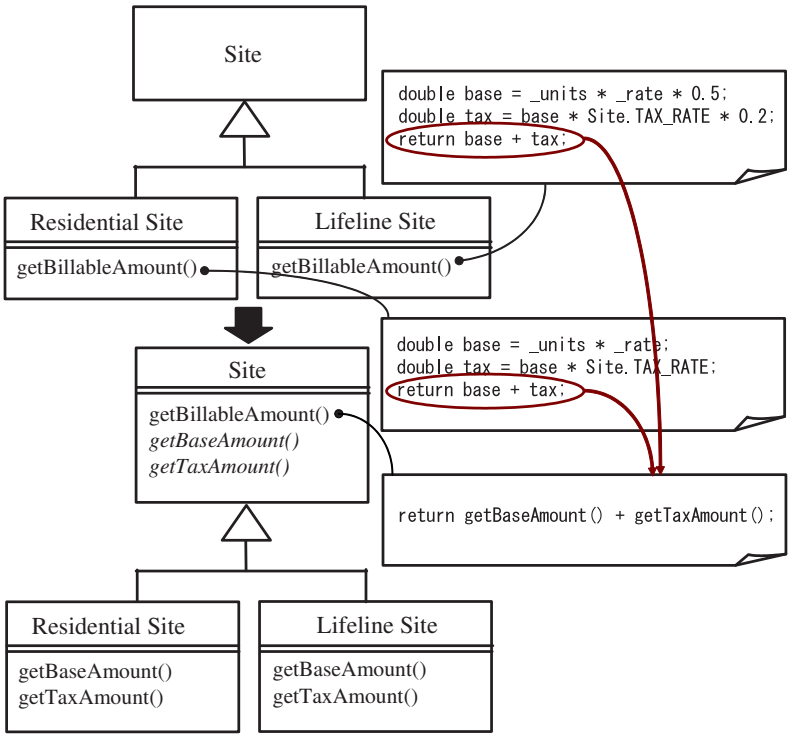


図 4: リファクタリング (テンプレートメソッドの形成)

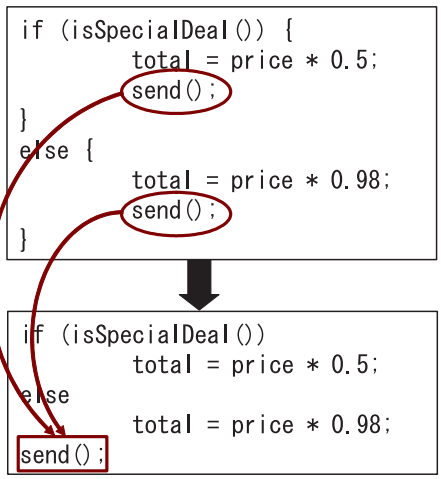


図 5: リファクタリング (重複した条件記述のコード片の統合)

2.2 コードクローンと既存のコードクローン検出法

コードクローンとは，ソースコード中に含まれる同一もしくは類似したコードのことであり，いわゆる “重複したコード” のことである．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 [10][28].

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．

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．

偶然

偶然に，開発者が同一のコード片を書い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．

もしコードクローンが存在した場合には，2.1.2 で述べたように，一般的にコードの変更等が困難であると言われ，保守容易性低下の一因となる．このようなコードクローンによる問題に対処する方法としては，

- コードクローン情報の文書化を行うことで変更の一貫性を保つ，
- コードクローンを自動で検出する

の2つがある [23]．しかし，コードクローン情報の文書化には全てのコードクローンに対する情報を常に最新に保つことに非常に手間がかかるため，現実には困難である．そこで，これまでにさまざまなコードクローン検出手法やツールが提案されている [1][4][5][6][7][8][9][10][13][28][30][31][32][33]．それぞれの手法やツールの特徴は次のようになっている (我々の開発した CCFinder は 2.3 節参照)．

Covet

文献 [32] 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[10]

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]

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 [16] を用いているため線形時間で解析可能である．

Duploc[13]

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[33]

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 [30]

関数等にまとめるのに適したコードクローンを抽出を目的として，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．[30] 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 [31]

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[7][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[1]

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．

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 [11] も，CloneDR，Covet，JPlag，Moss，そして我々の開発した CCFinder を含めた 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．

2.3 コードクローン検出ツール CCFinder

2.3.1 定義

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 と β は互いにクローンであるという [23]．またペア（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，クローンの同値類をクローンクラスと呼ぶ．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．

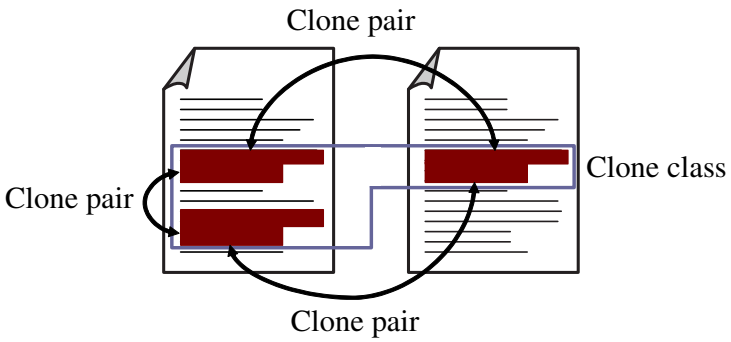


図 6: クローンペアとクローンクラス

2.3.2 概要

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 [23].

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．

2.3.3 コードクローン検出処理手順

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 4 つの過程から成り立っている．

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．

2.3.4 検出例

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 7 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 2 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 7 のソースコードに対しコードクローン検出を行うと，図 7 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．

```
    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 7: コードクローン検出例

- 名前空間の違い (e.g. “org.apache.regexp.RE” と “RE”).
- 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，2.3.2 節で述べた目的のため，CCFinder のトークン変換処理によって吸収されている．

2.4 コードクローン分析環境 Gemini

Gemini は，内部的に CCFinder を実行し，CCFinder から得られた解析結果を基に分析環境を提供する．システムの構成を図 8 に示す．Gemini は主に 5 つのコンポーネント:

1. コードクローン検出部（Code clone detector），
2. クローンペア管理部（Clone pair manager），

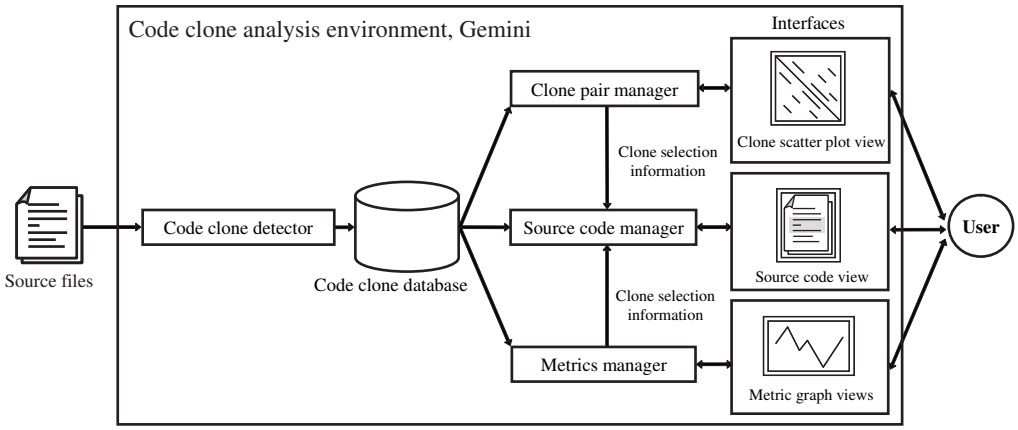


図 8: Gemini の構成

3. メトリクス管理部（Metrics manager），
4. ソースコード管理部（Source code manager），
5. ユーザインターフェース，

で構成されている．まず始めに，コードクローン検出部にソースファイルが入力され，コードクローンを検出する．次に，クローンペア管理部と，メトリクス管理部がその解析結果であるコードクローン情報を各インターフェースを通して視覚化する．それらのインターフェース上では，ユーザは任意のクローンペア (あるいは，クローンクラス) を選択することができ，その選択によって，実際のソースコードをソースコード管理部とそのインターフェースを通して参照することができる．

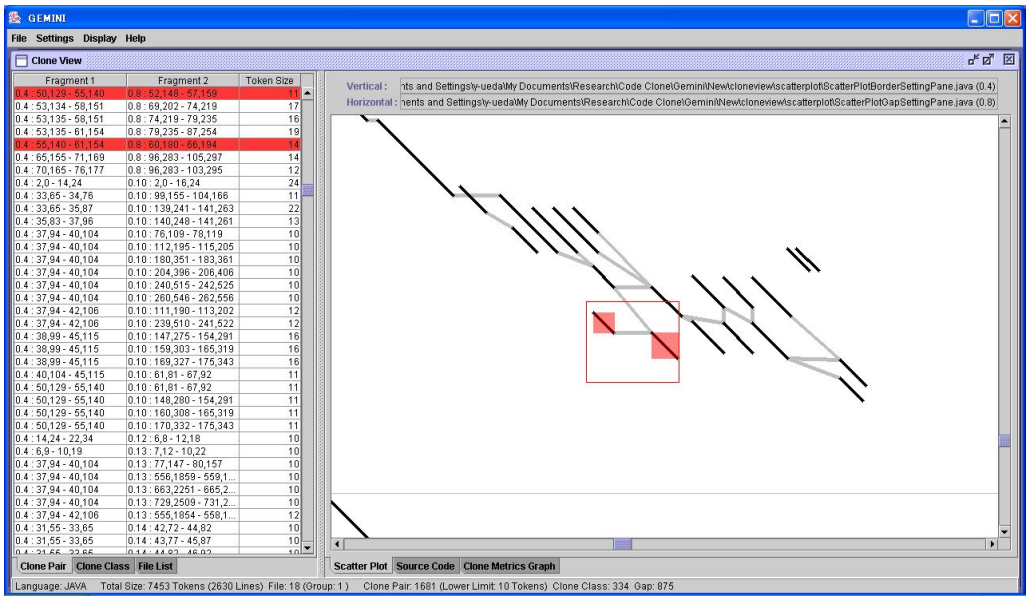
次節から図 8 における各コンポーネントについて順に述べる．

2.4.1 コードクローン検出部

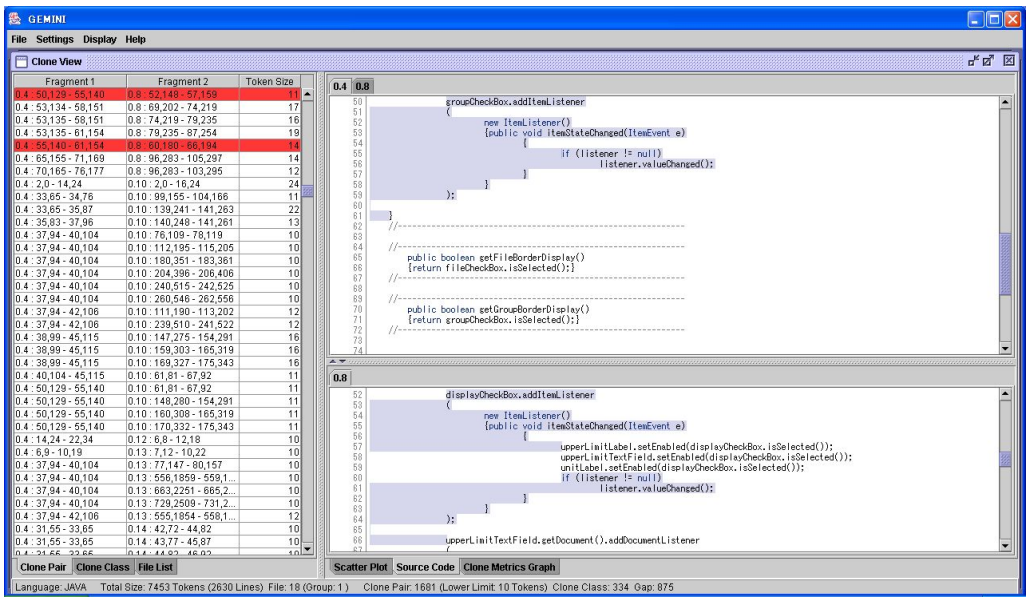
解析対象のファイルや，コードクローン検出のパラメータを管理する．CCFinder を利用して検出したコードクローンの位置情報も管理する．

2.4.2 クローンペア管理部

クローンペア位置情報を基にして，利用者の要求に応じて，クローン散布図を表示する．クローン散布図上での GUI による操作として，拡大・縮小，および，任意のクローンペア集合を「選択状態」にすること，がある (選択状態は，後述する他のサブシステムと連携した操作で使われる)．図 9(a) に選択状態のクローンペア集合を含んだクローン散布図の例を示す．矩形で囲まれたクローンペアが選択状態になっている．クローン散布図はソースコー



(a) クローン散布図



(b) 選択されたクローンのソースコード

図 9: Gemini スナップショット

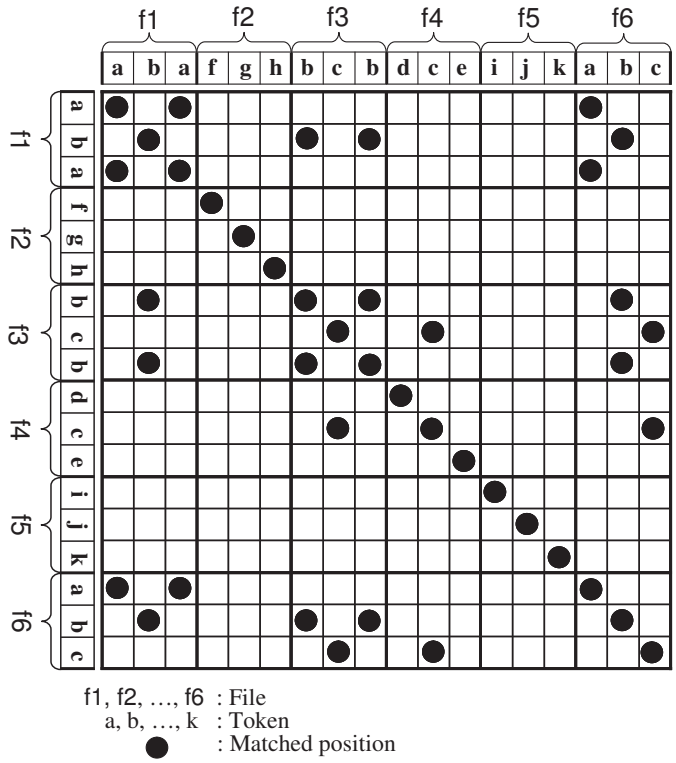
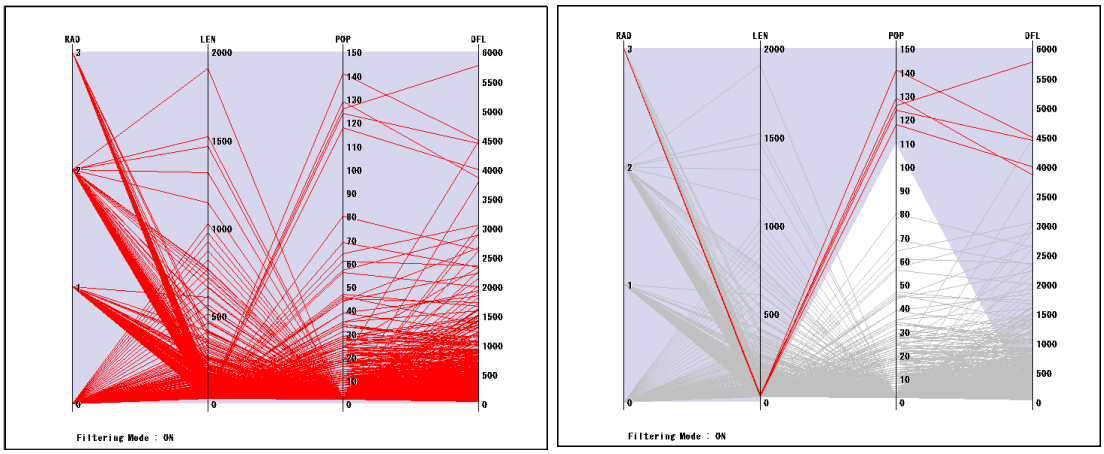


図 10: クローン散布図モデル

ドのどの部分にクローンペアが存在するのかを示す図である．一目でソースコード中のコードクローンの分布状況がわかるので，コードクローン解析の初期段階では非常に有効な解析手段となりうる．

クローン散布図

Gemini が表示するクローン散布図の簡単なモデルを図 10 に示す．散布図の原点は左上隅にあり，水平軸，垂直軸は，それぞれソースファイルの並び（f 1 から f 6）に対応している．両軸上で，原点から順に，それぞれのソースファイルに含まれるトークンが並んでいる．座標平面内に点がプロットされている部分は，その両軸の対応するトークンが一致することを意味する．したがって，散布図の主対角線は，両軸の同じ位置のトークンを比較することになり，すべての点がプロットされることになる．また，点の分布は主対角線に対して線対称となる．一定の長さ (CCFinder で設定される最小一致トークン数) 以上の対角線分が，検出されたクローンペアである．図 10 では，`f1` から `f6` までのファイルが，それぞれ 3 つのトークンを含んでいる．ファイルの並びはファイル名の辞書順となっている．検出されるクローンペアは `f1` と `f6` に含まれる “ab” というトークン列と，`f3` と `f6` に含まれる “bc” というトークン列である．



(a) 全選択

(b) $POP(C)$ が 100 以上のクローンクラスのみ選択

図 11: メトリクスグラフスナップショット

2.4.3 メトリクス管理部

クローンペア位置情報から，4 種のメトリクスを算出する．利用者の要求に応じて，メトリクスグラフによってメトリクス計測値を表示する．図 11 はメトリクスグラフのスナップショットである．メトリクスグラフは多次元並行座標表現 [29] を用いている．Gemini で用いている並行軸は $RAD(C)$,$LEN(C)$,$POP(C)$,$DFL(C)$ の 4 種類である．

$RAD(C)$: $RAD(C)$(Radius of clone-class) は, クローンクラス C 内のコードフラグメントが含まれるファイル集合 F が，ファイルシステムの中でディレクトリ構造的にどれだけ分散しているかを表すメトリクスである．ディレクトリ構造を表す木構造を考え，F 内の全てのファイルに共通の親ノードの中で最も下位層に存在するノードまでの距離を求め,C 内でのその最大値を $RAD(C)$ として定義する．

$RAD(C)$ の値が大きいならば, クローンクラス C 内のクローンは，システム内において広範に分布し，バグの作り込まれたクローンに修正を施すのはより困難であると考えられる．

$LEN(C)$: $LEN(C)$(Length) は, クローンクラス C 内に存在する最もトークン数の多いコード片のトークン数である．

$POP(C)$: $POP(C)$(Population of clone-class) は，クローンクラス C 内のコード片単位の要素数である．$POP(C)$ が高いということは，より多くの箇所にクローンが分散して

いることになる．

$DFL(C)$: $DFL(C)$(Deflation by clone-class) とは，クローンクラス C に含まれるコード片に共通するロジックを実装するサブルーチンを作り，各コード片をそのサブルーチンの呼び出しに置き換えた場合の減少が予測されるトークン数である．

新たに追加されるコードはサブルーチン実装コードと，サブルーチン呼び出しコードであるが，サブルーチンサイズは $LEN(C)$ トークン，呼び出しコードは全て併せて $5 \times POP(C)$ トークンとなる．呼び出し回数の $POP(C)$ に 5 をかけているのは，1 回の呼び出しにかかるトークン数は「サブルーチン名，(，引数，)，;」の 5 トークンとみなしているためである．従って $DFL(C)$ は以下の式によって定義することができる．

$$\begin{aligned} DFL(C) &= (\text{ 再構築対象コードサイズ }) - (\text{ 再構築後コードサイズ }) \\ &= (LEN(C) \times POP(C)) - (LEN(C) + 5 \times POP(C)) \\ &= (LEN(C) - 5) \times (POP(C) - 1) - 5 \end{aligned}$$

これら 4 つのメトリクスは図 11 の縦軸となっており，それぞれにメトリクス名のラベルがついているのがみてとれる．このグラフにおいて，各クローンクラス毎に 4 つの軸棒の点を結ぶ折れ線が引かれている．ユーザはこれら 4 つの座標の上限と下限を変更することで任意のクローンクラスを選択することが可能である．例として図 11(b) は $POP(C)$ の値がが 100 以上のクローンクラスのみを選択した状態を示している．

2.4.4 ソースコード管理部

指定されたソースファイルの内容をソースコードビューを通じて表示する (図 9(b) 参照).

2.4.5 サブシステム間の連携

クローンペア管理部，ソースコード管理部，メトリクス管理部の間で「選択状態」を用いた連携が可能になっている．この連携により，利用者は例えば (1) クローン散布図でクローンペアを選択した後，そのソースコードを表示する，(2) メトリクスグラフで一部のクローンクラスを選択した後，そのソースコードを表示する，(3) メトリクスグラフで長い (LEN が大きい) クローンクラスだけを選択した後，クローン散布図ではそれらがどこにあるのか調べる，といった対話的操作を行うことができる．例えば，図 9(b) は，図 9(a) において選択されたコードクローンのソースコードを表示している．

2.5 Gapped Clone

コピーアンドペーストによる再利用を考えた場合，ソフトウェア開発者は普通，コピーした部分そのままを使うことはあまりない．コピー後に部分的な修正を行なう場合がほとんど

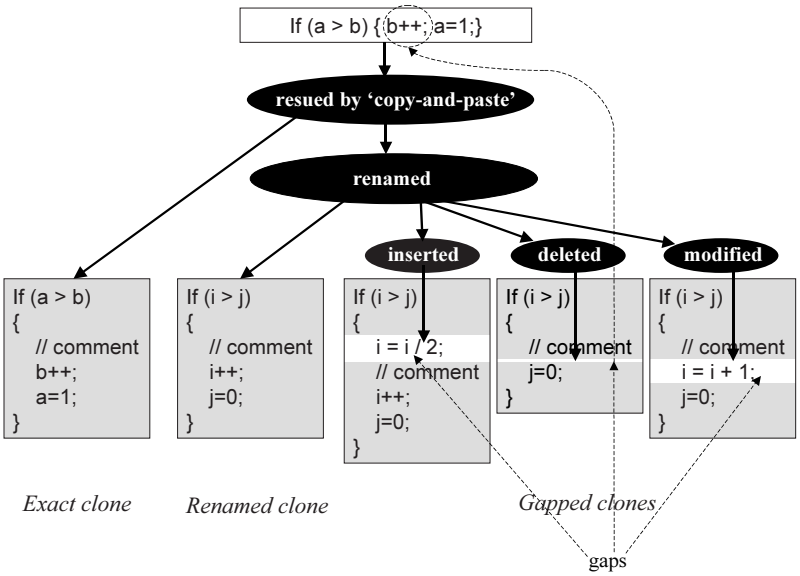


図 12: クローンの分類

である．部分的な修正といっても変数名，関数名などの置換だけでなく，文を挿入，削除するなどといったことも行なわれる．このようなことから，コピー元のコード片とコピー先のコード片では若干の違いが生じることとなる．この違いをギャップといい，この違いをもったコードクローンをギャップドクローンと呼ぶ．再利用という視点から考えた場合，コードクローンをは図 12 のように分類ができると我々は考えた．しかし，これらの分類のうち，CCFinder が実際に検出できるのは Exact clone と Renamed clone の二種類のみである．

開発者が実際にコード片を比較した時は，些細な修正が入ってようとも，それらがギャップドクローンであることは容易に判断することができる．しかし CCFinder はギャップドクローンを複数の小さなクローンとして検出してしまい，1 つのコードクローンとして検出することができないのである．また CCFinder の検出するコードクローンの最小トークン数はユーザによって予め指定されていなければならないのであるが，この値の設定次第では，小さいコードクローンを検出することすら不可能なのである．それならば，検出する最小トークン数を小さくすればいいとなるのであるが，そのようなことをした場合，非常に多くの小さなコードクローンが発見されてしまい，実用上意味をなさない．文献 [38] ではこれらの問題を解決するための提案手法を示している．簡単に説明すると，Exact Clone と Renamed Clone とそれらの間にあるギャップを一連のクローンとみなすのである．

2.6 コードクローンの利用目的

ソースコードからコードクローンを検出する目的としては，

1. ソースコード中のコードクローンの量，分布状態などを把握し，そのソフトウェアの品質・複雑度の調査・予測，
2. ソフトウェアの剽窃の調査，
3. 検出したコードクローンの除去，

などがあげられる．1，2 の場合はソースコード中の最大長のコードクローンを検出することが望ましいのに対し，3 を目的とした場合は，集約の容易なコードクローンを検出することが重要となってくる．

これまでに，集約を目的としたコードクローン検出はいくつか行なわれている．例えばプログラム依存グラフを用いる手法 [30][31] が提案されている．この手法は，ソースコードからプログラム依存グラフを構築し，そのグラフ上でのコードクローン検出を行なうことにより，高精度の検出を可能としている．しかし，プログラム依存グラフ構築の時間コストは非常に大きく，実際にソフトウェアの開発現場で適用するのは難しい．また，Mayland ら [32] は特徴メトリクスを用いたコードクローン検出を行なっている．彼らはコードクローン検出対象を，関数・メソッド単位に絞り込み，各関数・メソッドに対して 21 種類のメトリクス値を計測し，それらを定量的に特徴づける．そして，そのメトリクス値の比較により 8 段階の類似度を用いてコードクローンを検出している．この手法では，コードクローン検出対象を，関数・メソッドに特定しているため，関数・メソッド内部に存在するコードクローンの検出，集約を行なうことは不可能である．

これらの問題を解決するコードクローン検出手法として，本研究では言語における構造的なまとまりをもったコードクローンを検出する手法を提案する．また，リファクタリングを目的としていることから，集約方法の補助を目的としたメトリクスを提案し，検出したコードクローンを定量的に特徴づける．

3　リファクタリングを目的としたコードクローン解析手法

本研究では，以下に示す 2 段階のコードクローン検出手法を提案する．

1. プログラムのソースコードから，言語における構造的なまとまりを持ったコードクローンを検出する．
2. 検出したコードクローンに対してメトリクスを用いた定量的特徴づけを行ない集約方法を提示する．

コード片 1

```
609:    reset();
610:    grammar = g;
611:    // Lookup make-switch threshold in the grammar generic options
612:    if (grammar.hasOption("codeGenMakeSwitchThreshold")) {
613:        try {
614:            makeSwitchThreshold = grammar.getIntegerOption("codeGenMakeSwitchThreshold");
615:            //System.out.println("setting codeGenMakeSwitchThreshold to " + makeSwitchThreshold);
616:        } catch (NumberFormatException e) {
617:            tool.error(
618:                "option 'codeGenMakeSwitchThreshold' must be an integer",
619:                grammar.getClassName(),
620:                grammar.getOption("codeGenMakeSwitchThreshold").getLine()
621:            );
622:        }
623:    }
624:
625:    // Lookup bitset-test threshold in the grammar generic options
626:    if (grammar.hasOption("codeGenBitsetTestThreshold")) {
627:        try {
628:            bitsetTestThreshold = grammar.getIntegerOption("codeGenBitsetTestThreshold");
```

コード片 2

```
623:    }
624:
625:    // Lookup bitset-test threshold in the grammar generic options
626:    if (grammar.hasOption("codeGenBitsetTestThreshold")) {
627:        try {
628:            bitsetTestThreshold = grammar.getIntegerOption("codeGenBitsetTestThreshold");
629:            //System.out.println("setting codeGenBitsetTestThreshold to " + bitsetTestThreshold);
630:        } catch (NumberFormatException e) {
631:            tool.error(
632:                "option 'codeGenBitsetTestThreshold' must be an integer",
633:                grammar.getClassName(),
634:                grammar.getOption("codeGenBitsetTestThreshold").getLine()
635:            );
636:        }
637:    }
638:
639:    // Lookup debug code-gen in the grammar generic options
640:    if (grammar.hasOption("codeGenDebug")) {
641:        Token t = grammar.getOption("codeGenDebug");
642:        if (t.getText().equals("true")) {
```

図 13: CCFinder の検出したコードクローン例 1

3.1 言語における構造的なまとまりを持ったコードクローンの検出

まず，言語における構造的なまとまりを持ったコードクローン検出手法を提案する [17][18][19][20][21]. この過程では，既存のコードクローン検出ツール CCFinder を用いる．CCFinder は 2.3 節で説明したように，ソースコードをトークン単位に分割し，そのトークン列に対してサフィックスツリーアルゴリズム用いてコードクローン検出を行なう．CCFinder の解析は非常に高速であるが，検出されるコードクローンは単に最大一致トークン数によるものであり，コードクローンの凝集度は考慮されていない．そこで，CCFinder の検出したコードクローン内に存在する最大長の構造的なまとまりを抽出する．実際の抽出過程は以下の 3 つからなる．

1. 対象のソフトウェアに対し CCFinder を実行し，コードクローンを検出する．
2. 次に，対象のソフトウェアを解析し，構造的なまとまりをもったブロックの位置情報を抽出する．例えば，Java プログラムを対象とした場合は，構造的なまとまりをもったブロックとは具体的には以下で示すものである．

   宣言　：class { }，interface { }
   メソッド　：メソッド，コンストラクタ
   文　：if 文，for 文，while 文，do 文，switch 文，try 文，synchronized 文，static 文
3. 最後に，コードクローンの情報と，ブロックの位置情報を突き合わせ，ユーザに指定された最小トークン数以上のコードクローンを抽出する．

図 13，14 は CCFinder が検出したコードクローンを示している．図 13 のコード片 1 とコード片 2，図 14 のコード片 3 とコード片 4 の強調表示されている部分がそれぞれコードクローンとなっている．図 13 の場合，リファクタリングを行なうに際しては 612 行目から 623 行目までを集約することが適切であり，624 行目から 626 行目は集約することは困難である．本節で提案した抽出手法ではこの適した部分のみ（コード片 1 の 612 行目から 623 行目，コード片 2 の 626 行目から 637 行目）を抽出することが可能である．また，図 14 のコード片 3 とコード片 4 は CCFinder によってコードクローンとして検出されている（コード片 3 の 1010 行目 1016 行目とコード片 4 の 1530 行目から 1536 行目）が，このコードクローン中には特に構造的なまとまりは存在せず，集約することは困難である．提案手法ではこのようなコードクローンを識別することで，リファクタリングの候補から外すことができる．

CCFinder のコードクローン検出過程では，ソースコードに含まれるトークン数を n，検出された最も長いクローンのトークン数を t とした場合，$O(nt)$ 時間で解析結果を得ることが可能である，詳しくは文献 [28] を参照していただきたい．CCFinder の検出したコードクローンから構造的にまとまりのある部分を抽出する解析では，$O(n)$ の処理時間で対象ソースコードを構文解析しソースコード中の構造的なまとまりのある部分を抽出できる．また

コード片3

```
1007:    if ( inputState.guessing==0 ) {
1008:        buf.append(a.getText());
1009:    }
1010:    {
1011:    _loop144:
1012:        do {
1013:            if ((LA(1)==WILDCARD)) {
1014:                match(WILDCARD);
1015:                a=id();
1016:                if ( inputState.guessing==0 ) {
1017:                    buf.append('.'); buf.append(a.getText());
1018:                }
1019:            }
```

コード片4

```
1527:    if ( inputState.guessing==0 ) {
1528:        t=a.getText();
1529:    }
1530:    {
1531:    _loop84:
1532:        do {
1533:            if ((LA(1)==COMMA)) {
1534:                match(COMMA);
1535:                id();
1536:                if ( inputState.guessing==0 ) {
1537:                    t+=","+b.getText();
1538:                }
1539:            }
```

図 14: CCFinder の検出したコードクローン例 2

$O(cs\ log\ c)$（c は 1 つのファイル当たりに含まれるコードクローンの数，s は対象ソースコードの数，c，s いずれも n に対して非常に小さい値となる）の処理時間で CCFinder の検出したコードクローン情報と，ソースコード中の構造的なまとまりのある部分から，リファクタリングの容易なコードクローンを抽出できる．

他のアプローチ，例えばプログラム依存グラフを用いる手法 [30][31] と比較した場合，プログラム依存グラフ構築の時間コストが $O(m^2)$（m はソースコード中の文や式の数，m は n と比例関係にある）である [25] ことを考慮すると，この提案手法では，対象が大規模なソフトウェアであっても実用的な時間でリファクタリングの容易なコードクローンを検出できると期待できる．

3.2 抽出したコードクローンの分類

次に，抽出したコードクローンを意味解析しその解析結果をメトリクスとして数値化することによって，コードクローンを定量的に特徴づける．そして，このメトリクス値に基づき各コードクローンがどのようにリファクタリングできるのかをユーザに通知する．

コードクローンの除去手法には既存のリファクタリングパターン，特に「メソッドの抽出」と「メソッドの引き上げ」を用いる．

2.1.2 節で述べたように,「メソッドの抽出」とは，ソースコードのある一部分を新たなメソッドして再定義することである [15]．本来は長過ぎるメソッドや，複雑な処理の一部分に対して適用されるリファクタリングパターンであるが，抽出すべき箇所がクローンとなっていた場合，それらを共通のメソッドととして抽出することにより，集約することが可能である．図 1 はメソッドの抽出の一例を示している．この例ではメソッド printOwing 内に存在する 2 つの System.out.println 文が新たなメソッド printDetails として再定義されている．もしこのような System.out.println 文が他の場所にも存在した場合，このような抽出を行なうことによってコードクローンの集約が可能である.「メソッドの引き上げ」とは，ある親子

クラス関係が存在した場合に，子クラスに存在するメソッドを親クラスに引き上げることである [15]．子クラスに存在するメソッドを親クラスに引き上げるにはいろいろな理由が考えられるが，もし共通の親クラスを持つ複数の子クラスにコードクローンとなっているメソッドが存在した場合は，それらを共通の親クラスに引き上げることによってコードクローンを集約することが可能である．図 2 は「メソッドの引き上げ」の例を示している．この例では Employee クラスを継承している Salesman クラスと Engineer クラスに共通して getName() というメソッドが存在している．もし 2 つのクラス中に存在するこのメソッドがクローンとなっていた場合は，それらを引き上げることによって，クローンの集約が可能である．

抽出したコードクローンに対してメトリクス値を計測することによって，これら 2 つのリファクタリングパターンが適用可能なコードクローンの絞り込みを可能にする．まず,「メソッドの抽出」を適用するための条件として以下の 3 つの条件があげられる．

1. 文単位のコードクローンである，
2. 全てのコードクローンがある 1 つのクラス内に存在する，
3. コードクローンの内側では，コードクローンの外側で定義された変数を高々1 つしか使用（代入，参照）していない．

「メソッドの抽出」を行なう単位は図 1 でも示したように，既存のメソッド内部の一部である．よって，構造的なまとまりを持つコードクローンを対象とした場合は，文単位のコードクローンが対象となる（条件 1）．また，抽出部分は同クラス内の private メソッドとするので，全てのコードクローンがある 1 つクラス内に存在することが望ましい（条件 2）．全てのコードクローンが同クラス内に存在しなくても，public 修飾子を用いたり，親クラスにメソッドを抽出することで全ての子クラスからのアクセスを許可することは可能であるが,「メソッドの抽出」とは本来抽出したメソッドを抽出元と同クラス内に再定義するものであり，条件 2 を必要としている．またコードクローンの内側で，コードクローンの外側で定義された変数を使用している場合は,「メソッドの抽出」を行なう際にその外部定義変数を引数として与える必要がある．さらにコードクローン内での変更を確実にメソッドの呼出元に反映させるために返り値として返す．厳密にいえば，コードクローンの外部で定義された変数がインスタンス変数の場合は，抽出したメソッド内で一貫して同じメモリ領域を指している限りその変数が指すオブジェクトへの変更はメソッドの外部でも反映されているので，必ずしも返り値として返す必要はなく「メソッドの抽出」を行なうことが可能である（図 15 参照）．しかし，将来的にバグ修正や機能追加などでメソッドのコードが変更された場合，外部定義の変数が同じメモリ領域を指し続けている保証はなく，もし同じメモリ領域を指さなくなった場合はバグとなってしまうのでそのようなコードは保守性の面からみて問題があると考えられ，外部定義の変数を高々1 つ用いているコードクローンを対象としている．

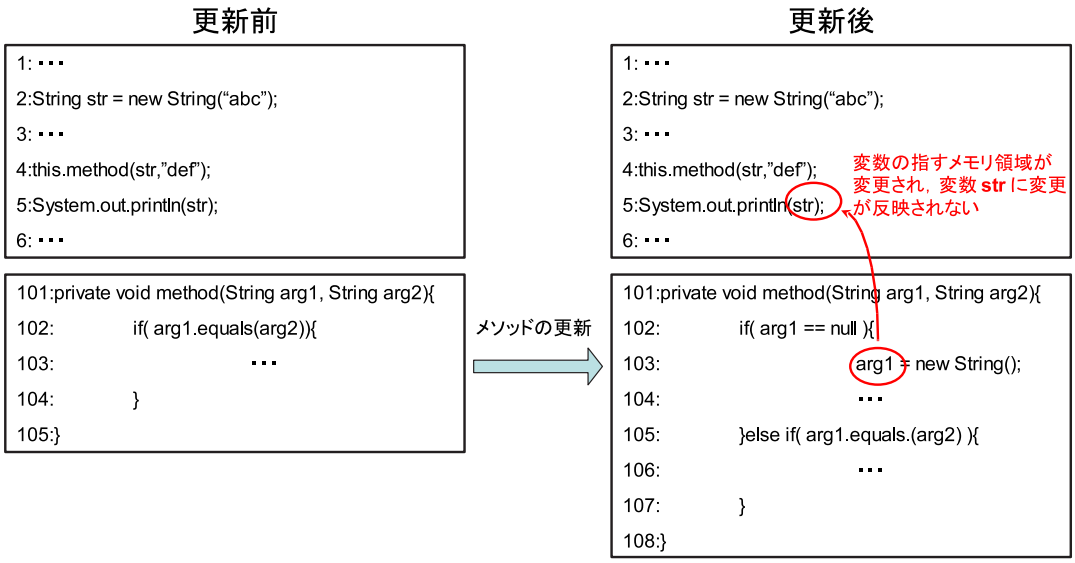


図 15: 「メソッドの抽出」が引き起こすバグ

一方,「メソッドの引き上げ」を行なう条件としては以下の 2 つがあげられる.

1. メソッド単位のコードクローンである,
2. コードクローンが存在するクラスが共通を親クラスを持つ.

「メソッドの引き上げ」とは既存のメソッドを親クラスに引き上げることであるので，対象となるコードクローンの単位はメソッドとなる．また全てのコードクローンを同一のクラスに引き上げるので，コードクローンが存在するクラスは引き上げ先となる共通の親クラスを持っていなければならない.

従って,「メソッドの抽出」と「メソッドの引き上げ」を適用できるコードクローンを抽出するために以下の 2 つのことがらについて解析を行う必要がある.

1. コードクローンの内部でのコードクローンの外部で定義された変数の使用数（代入，参照数）,
2. コードクローン（クローンクラス）のクラス階層における分散度.

1 については 2 つのメトリクス $RVK(C)$，$RVN(C)$ を定義する．$RVK(C)$ とはコードクローンの内部において使用しているコードクローンの外部で定義された変数の数を表している．また $RVN(C)$ は $RVK(C)$ に使用回数の重みをつけたものである．例えば，図 16 が if 文単位のコードクローンであると考える．このコード片では，外部定義の変数 t，location

```
if (t instanceof BuildException) {
        BuildException be = (BuildException) t;
        if (be.getLocation() == Location.UNKNOWN_LOCATION) {
                be.setLocation(location);
        }
        throw be;
}
```

図 16: $RVK(C)$，$RVN(C)$ の例

を用いている（Location はクラス名）ので $RVK(C)$ の値は 2 となる．また，変数 t の使用回数は 2，変数 location の使用回数は 1 であるので，$RVN(C)$ の値は 3 となる．

2 についてはメトリクス $DCH(C)$ を定義する．$DCH(C)$ はそのクローンクラスに属するコード片のクラス階層における分散度を整数値で表す．分散度が高いほど $DCH(C)$ の値も大きくなる．具体的には，全てのコードクローンがある 1 つのクラス内に存在する場合は $DCH(C)$ の値は 0，あるクラスとその子クラスに存在する場合は 1，と共通の親クラスまでの距離を整数値で表す．共通の親クラスを持たない場合は-1 とする．また，共通の親クラスを持つ場合であっても，もしその共通の親クラスが解析対象となっているソフトウェアに含まれるクラスではなく JDK のようなライブラリに含まれるクラスであった場合は，そのクラスは親クラスとはみなさない．なぜなら，ライブラリのソースコードを修正することは現実的ではないため，そのような親クラスには子クラスに定義されたメソッドを引き上げることはできないからである．

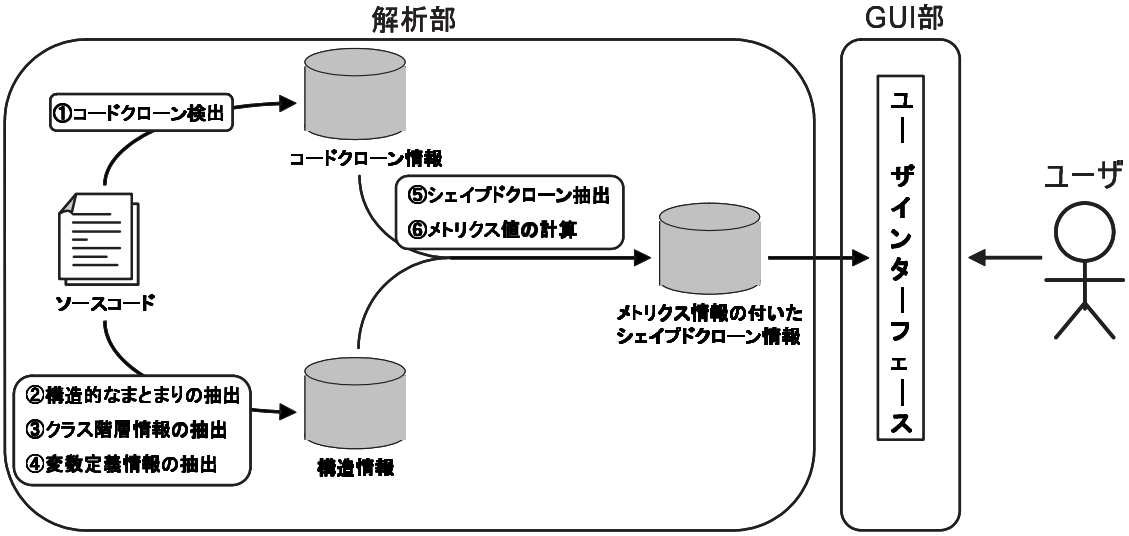


図 17: Cancer による解析の流れ

4 リファクタリング支援ツール：Cancer

4.1 システム概要

本リファクタリング支援ツール Cancer は Java で実装されており，JDK1.4 以上の VM が実行可能な環境で動作する．図 17 は Cancer の解析の流れを示している．Cancer は大きく分けて二つのコンポーネント，解析部と GUI 部に分けられる．解析部では大きく分けて以下の 6 つの解析を行なっている．

1. コードクローン検出,
2. 構造的なまとまりの抽出,
3. クラス階層情報の抽出,
4. 変数定義情報の抽出,
5. 構造的なクローン抽出,
6. メトリクス値の計算.

このうち，コードクローン検出は内部的に CCFinder を実行して行なっている．その他の解析のうち構文，意味解析の必要な部分はオープンソースの構文解析器生成ツール JavaCC[24] を用いて作成した．また解析部で抽出した「メトリクス情報の付いた構造的なドクローン情報」は XML 形式で GUI 部に渡される．GUI 部では解析部から渡されてた XML ファイルを読み込み，グラフ，表などの形式を用いてユーザに示す．図 18，19 は Cancer の実行画面

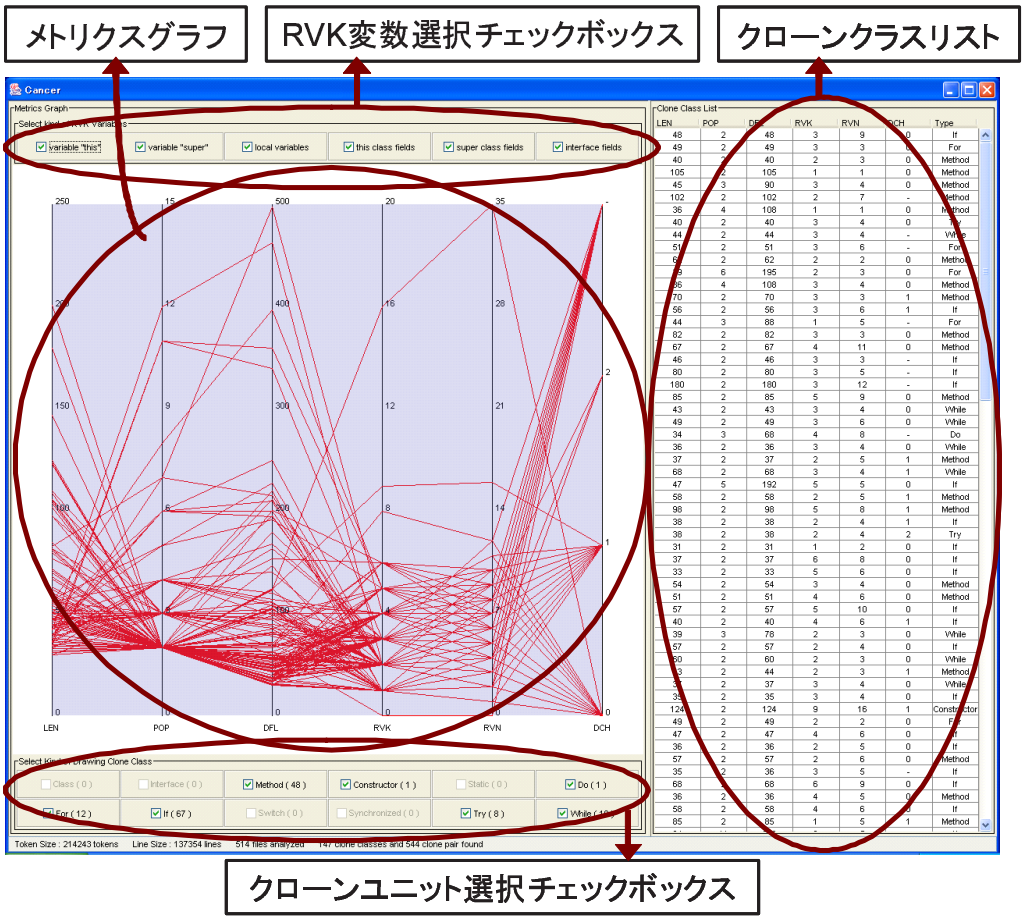


図 18: Cancer スナップショット 1

のスナップショット（GUI 部）であり，各 GUI コンポーネントが何であるかの注釈が付けられている．

4.2 各コンポーネントの働き

本節では Cancer の各ユーザインターフェースを簡単に説明する．

4.2.1 メトリクスグラフ

Gemini ではメトリクスグラフで用いるメトリクスは $RAD(C)$，$LEN(C)$，$POP(C)$，$DFL(C)$ の 4 つであったが，Cancer では 3 章で提案した 3 つのメトリクス，$RVK(C)$，$RVN(C)$，$DCH(C)$ と Gemini で用いている $LEN(C)$，$POP(C)$，$DFL(C)$ の合わせて 6 つのメトリクスを用いる．本ツールのメトリクスグラフは Gemini のメトリクスグラフと同様に各メトリクスにつき上限，下限を設定することで任意のクローンクラスを選択可能であ

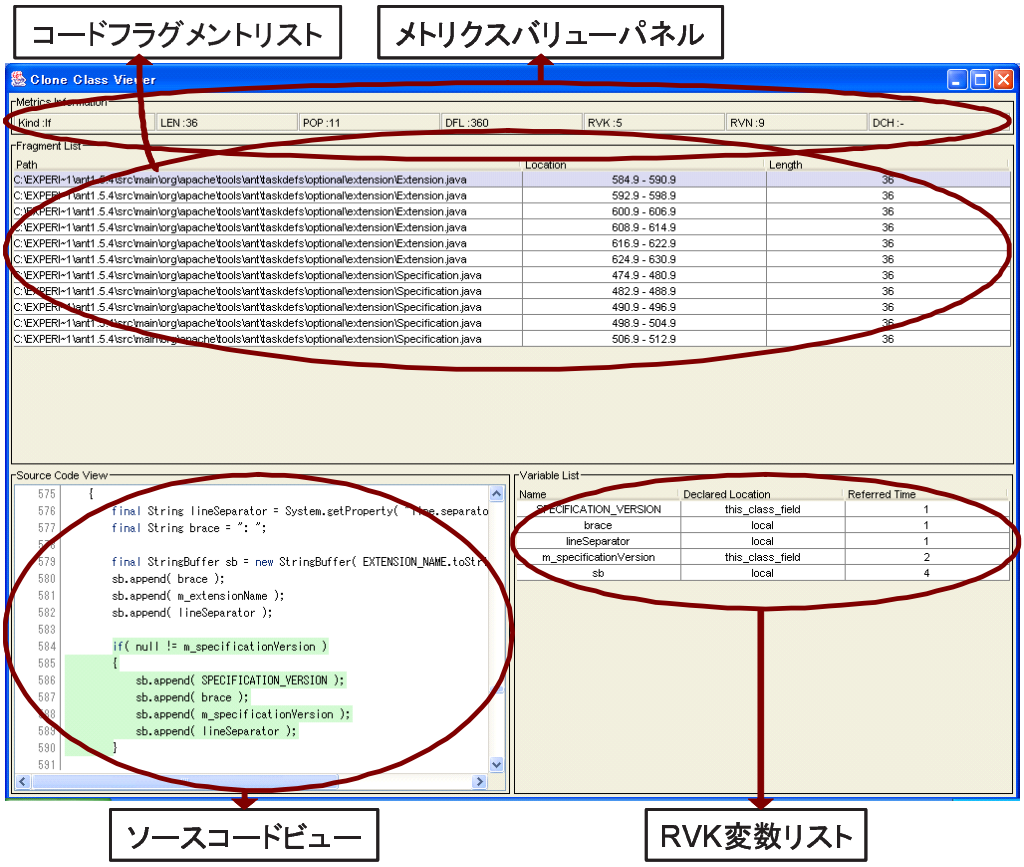


図 19: Cancer スナップショット 2

る．選択結果はクローンクラスリスト（4.2.4 節）に反映される．

4.2.2 RVK 変数選択チェックボックス

RVK 変数選択チェックボックスでは RVK 変数としてカウントする変数の種類を選択する．現在 RVK 変数の種類としては，現在のところ以下の 6 つとなっている．

- 自クラスのフィールド変数,
- 親クラスのフィールド変数,
- インターフェースのフィールド変数,
- 変数 “this”,
- 変数 “super”,
- ローカル変数.

ユーザはこの 6 つの変数それぞれについて，RVK 変数としてカウントするかどうかをチェックボックスを用いて選択できるようになっている．例えば，1 つのクラス内で閉じたリファクタリングを行なう場合は，自クラスのフィールド変数や，親クラスのフィールド変数，インターフェースのフィールド変数などは同じクラス内なら同様にアクセス可能であるので $RVK(C)$ としてカウントする必要がない．一方,「メソッドの引き上げ」など，複数のクラスに跨ったリファクタリングを行なう場合は，自クラスのフィールド変数なども $RVK(C)$ としてカウントする必要がある．

4.2.3 クローンユニット選択チェックボックス

クローンユニット選択チェックボックスとは，コードクローンを 3.1 節で説明した 12 種類の分類に基づいて選択をするためのものである．クローンユニット選択チェックボックスでは対象とするコードクローンの単位の選択をする．現在は Java を解析対象としているので，コードクローンの単位は 3 章で説明した 12 種類である．例えば,「メソッドの引き上げ」の適用可能なコードクローンを抽出したい場合は,「メソッドの引き上げ」の適用可能なコードクローンの単位はメソッドのみであるので，メソッドのチェックボックスにチェックを入れ，他の 11 個のチェックボックスのチェックを外す．

4.2.4 クローンクラスリスト

クローンクラスリストにはメトリクスグラフにおいて選択状態にあるクローンクラスの一覧が表示される．また，このリストに表示されたクローンクラスは各メトリクス値の昇順，降順でソート可能である．このリストのクローンクラス上において，ダブルクリックをすることで，そのクローンクラスについてのより詳細な情報を表した図 19 のウィンドウが立ち上がる．

4.2.5 メトリクスバリューパネル

メトリクスバリューパネルにはそのクローンクラスのメトリクス値一覧が表示される．

4.2.6 コードフラグメントリスト

コードフラグメントリストにはそのクローンクラスに含まれるコード片の一覧が表示される．各コード片について，そのコード片が存在するファイルへのパス，ファイル内での位置情報（開始行，開始列，終了行，終了列）と，コード片のトークン数の 3 つの情報が表示されている．

4.2.7 ソースコードビュー

ソースコードビューでは，フラグメントリストにおいて選択されたコード片周辺のソースコードが表示される．コードクローンとなっている部分は強調して表示される．

4.2.8 RVK 変数リスト

RVK 変数リストには，フラグメントリストにおいて選択されたコード片における RVK 変数の一覧が表示される．各 RVK 変数について，変数名，種類，使用回数の 3 つの情報が表示される．

4.3 リファクタリング方法

本節では Cancer を用いて行なう「メソッドの抽出」と「メソッドの引き上げ」のリファクタリング方法について述べる．

「メソッドの引き上げ」の適用対象となるコードクローン（クローンクラス）の絞り込み条件は，

1. メソッド単位のコード片を要素としているクローンクラス，
2. $DCH(C)$ の値が 1 以上であるクローンクラス．

の 2 つである．「メソッドの引き上げ」とは既存のメソッドを親クラスに引き上げることであるから，対象となるコード片はメソッド単位となり，の条件 1 が必要である．また，対象とするメソッドが存在するクラスには共通の親クラスが存在する必要があるので条件 2 が必要となる．

一方，「メソッドの抽出」の適用対象となるコードクローン（クローンクラス）の絞り込み条件は，

1. 文単位のコード片を要素としているクローンクラス，
2. $DCH(C)$ の値が 0 であるクローンクラス．
3. $RVK(C)$ の値が 1 以下であるクローンクラス．

の 3 つである．「メソッドの抽出」とは既存のメソッド内のある部分を新たなメソッドとして抽出することであるので，抽出部分はメソッド単位よりも小さい必要がある．よって，条件 1 が必要となる．また，抽出によって新たに作成されたメソッドは，private メソッドとして定義し，同クラス内からのアクセスのみ許可するので条件 2 が必要となる．private メソッドとして宣言せずに，子クラスからのアクセスも許可することは可能であるが，このような場合はそれぞれの子クラスで「メソッドの抽出」を行ない，その後「メソッドの引き

上げ」を行なう必要があり,「メソッドの抽出」のみの適用とはならず，対象外としている．また，外部定義の変数は抽出したメソッドの引数として与え，返り値として返す必要があるので，$RVK(C)$ の値は 1 以下としている.「メソッドの抽出」は 1 つのクラス内での閉じたリファクタリングであるので，クラス内では制限なくアクセスできる自クラスのフィールド変数，親クラスのフィールド変数，インターフェースのフィールド変数，変数 “this”，変数 “super” の四種類の変数は $RVK(C)$ の対象外としている．厳密にいえば，$RVK(C)$ の値が 1 以下となっていなくても外部定義の変数がインスタンス変数の場合は，抽出したメソッド内で一貫して同じメモリ領域を指している限りその変数が指すオブジェクトへの変更はメソッドを抜けても反映されているので「メソッドの抽出」を行なうことが可能である（図 15 参照）．しかし，将来的にバグ修正や機能追加などでメソッドのコードが変更された場合，外部定義の変数が同じメモリ領域を指し続けている保証はなく，もし同じメモリ領域を指さなくなった場合はバグとなってしまうため，そのようなコードは保守性の面からみて問題があると考えられるので，$RVK(C)$ の値が 1 以下のコードクローンを対象としている．

5　評価

本節では，Cancer を用いて行なった適用実験について説明する．なお，この実験では CCFinder の検出するコードクローンの最小一致トークン数は 30 とした．この 30 という値は，これまで行なった数多くの CCFinder の適用事例から導き出された経験的な値である．また，Cancer の抽出する最小トークン数も 30 とした．この値については特に根拠はないが，今回は実験的に CCFinder の最小一致トークン数と同様の値を設定した．

5.1　学生プログラム 1

大阪大学大学院情報科学研究科 2 回生が作成したプログラムに対して適用実験を行なった．このプログラムは Java で作成されており，126 ファイル，約 16500 行である．このプログラムは一部はコード生成ツールを用いて作成されており，その部分には多くのコードクローンが検出されることが予想されるので，この適用実験ではコード生成ツールによって生成された部分を除いた 43 ファイル，約 8600 行に対してコードクローン検出を行なった．

検出を行なった結果,「メソッドの抽出」の条件を満たしているクローンクラスは 8 個であった．それぞれに対してソースコードを閲覧し，実際に「メソッドの抽出」が行なえるかを確認したところ，そのうちの 4 つに対しては行なうことが困難であることが判明した．図 20 は困難であったコードクローンのうちの 1 つのソースコードである．このクローンクラ

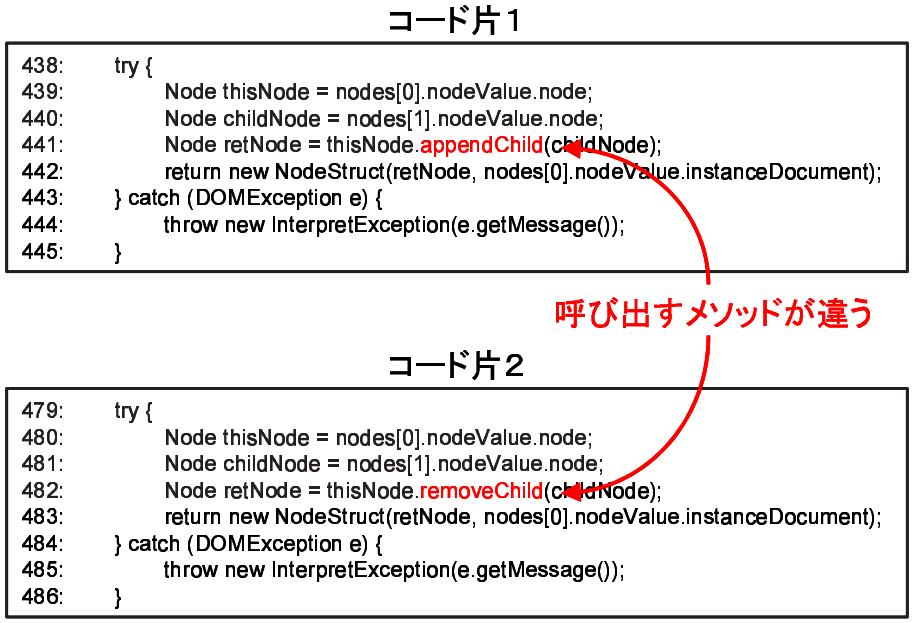


図 20: 学生プログラム 1：除去できなかったコードクローンの例

スの $POP(C)$ は 2 であり，try 文のコードクローンである．この 2 つのコード片は赤で示されている部分で異なるメソッドを呼び出していた．Cancer では $RVK(C)$，$RVN(C)$ の計算において，使用している変数がコードクローンの外部で定義されているのか，内部で定義されているのかの解析しか行なっておらず，変数が内部定義であった場合は，その変数に対してのメソッドの呼び出しは考慮していない．この解析不足により，実際には「メソッドの抽出」の適用が困難であるクローンクラスを 4 つ検出してしまっていた．

また,「メソッドの引き上げ」の条件を満たしているクローンクラスは 2 つであった．しかし，この 2 のクローンクラス共，内部定義の変数に対しての呼び出しメソッドの違いにより,「メソッドの引き上げ」の適用は困難であった．

5.2 学生プログラム 2

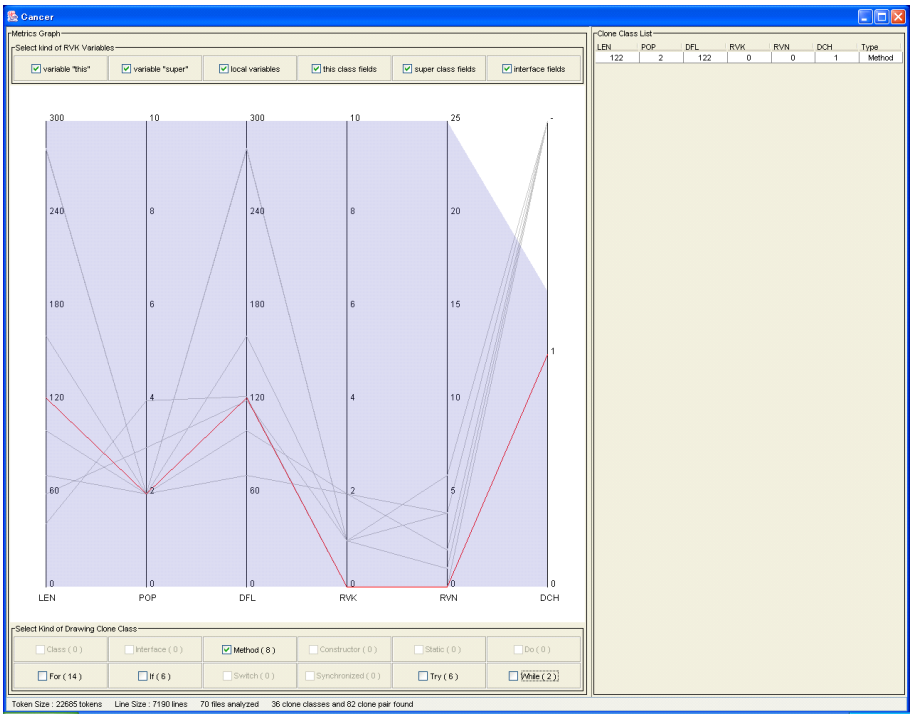
次に，大阪大学基礎工学部情報科学科 4 回生が作成したプログラムに対して適用実験を行なった．このプログラムは Java で作成されており，70 ファイル，約 7200 行である．

解析を行なった結果,「メソッドの抽出」の条件を満たしているコードクローンは 4 つ存在した．その内 3 つについては 5.1 節で除去できなかった場合と同様の状況であり，取り除くことが困難であることが予想される．残りの 1 つのクローンクラスについては，コードクローン内で用いている文字列定数が異なっているのみであり，この文字列定数を引数とすることで「メソッドの抽出」が適用可能である．

一方,「メソッドの引き上げ」の条件を満たしているコードクローンは 1 つ存在した．図 21(a) はその絞り込みの様子を表している．この図からわかるように，このコードクローンの $RVK(C)$ の値は 0，つまりこのメソッド単位のコードクローンは外部定義の変数を全く利用していないことがわかる．このコードクローンの $POP(C)$ の値は 2 であり，それぞれ別のクラス内のメソッドあった．また，$DCH(C)$ の値が 1 であることからもわかるように，このコードクローンが存在するクラスは共通のクラスを継承している（図 21(c) 参照）．図 21(b) はこのコードクローンのソースコードを表している．ソースコードからもわかるようにこのメソッドの内部は，引数として受け取った int 型の変数の値を調べ，その値に応じて int 型の値を返り値として返している．これらのことから，この 2 つのメソッドは「メソッドの抽出」を容易に適用可能である．

5.3 Ant

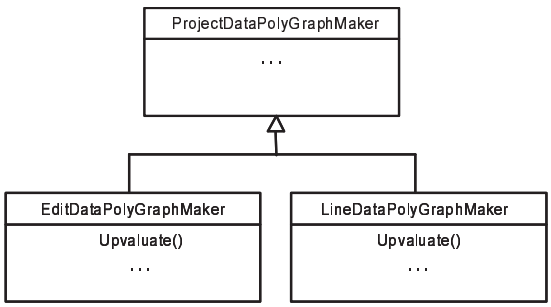
次に，オープンソースの Java ソフトウェアである Ant[2] に対してリファクタリングを行なった．Ant とは make のようなビルドツールの一種である．ビルド手順は XML で記述される．Ant のソースファイルは 627 個であり，総行数は約 18 万行である．



(a) 絞り込みの様子

```
private int upvaluate(int maxEdit){
    // 仮実装 上から2桁目を繰り上げ
    if(maxEdit < 10){
        return 10;
    }
    String str = String.valueOf(maxEdit);
    boolean needUpvaluate = false;
    for(int i = 1;i < str.length();i++){
        if(str.charAt(i) != '0'){
            needUpvaluate = true;
            break;
        }
    }
    if(needUpvaluate){
        int result = Integer.valueOf(str.substring(0,1)).intValue();
        result++;
        for(int i = 0;i < str.length()-1;i++) result *= 10;
        return result;
    }
    return maxEdit;
}
```

(b) コードクローンのソースコード



(c) クラス図から見たコードクローン

図 21: 学生プログラム 2

5.3.1 実験方法

Cancer が検出したコードクローンについて二種類のリファクタリングパターン,「メソッドの抽出」と「メソッドの引き上げ」の適用を試みる．この実験では 4.3 で述べた条件を

表 1: Cancer が抽出したコードクローンの数

	全体	メソッドの抽出	メソッドの引き上げ
全体	154	32	20
$POP(C)$ 3 以上	42	8	7

用いて絞り込みを行なった結果，リファクタリング可能と判断されたコードクローンの内，$POP(C)$ が 3 以上，つまり要素が 3 個以上存在するクローンクラスに対して除去を試みた．$POP(C)$ が 2 のものに関しては，実際のリファクタリングを行なう場合，2 箇所しかコードクローンになっていないものを集約することはあまりないと考えられ，対象外としている．

Ant に対して Cancer を実行したところ，154 個のクローンクラスを検出した．また，5.3.1 節で述べた「メソッドの抽出」と「メソッドの引き上げ」の条件を用いて検出したコードクローンを絞り込んだ結果のコードクローン数は表 1 に示す．つまりこの実験において，リファクタリングの対象となるクローンクラス数は，$POP(C)$ の値が 3 以上である「メソッドの抽出」もしくは「メソッドの引き上げ」の条件を満たしている 15 クラスとなる．

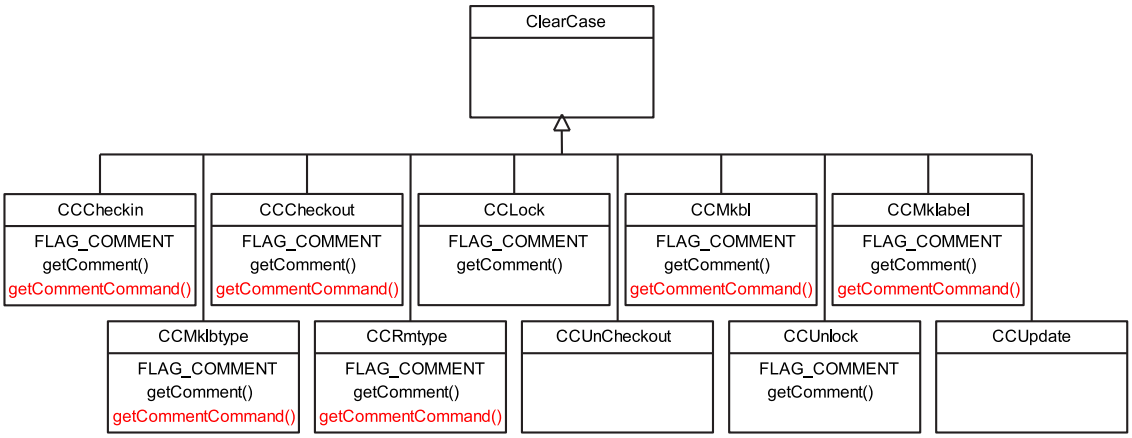
またリファクタリングを行なうことによって Ant の動作が変わっていないことを保証するために，1 つのクローンクラスを集約するごとに回帰テストを行なった．回帰テストには，Ant 付属のテストケース 220 個全てを用いた．このテストケースは JUnit[26] というユニットテストフレームワークを用いて記述されており，プロンプトに “./build.sh test” と入力するだけで全てのテストケースを実行させることができ，もしエラーが検出された場合はそこで中断する．Ant 付属の全テストケースを実行するのに用する時間は約 4 分であった．

5.3.2 リファクタリング

図 22 は「メソッドの引き上げ」の条件を用いて抽出した最も $POP(C)$ の値が大きかったクローンクラス（=6）のコードクローンを表している．このメソッド単位のコードクローン（図 22(a)）は図 22(b) に示すように，ClearCase を親クラスとする 10 個のクラスの内，6 個のクラスに存在していた．このメソッド単位のコードクローンは内部で getComment() というメソッドと，`FLAG_COMMENT` という変数を用いていた．メソッド getComment() は，このコードクローンが存在する全てのクラスにおいて定義されており，この呼び出し文は自クラスに存在する getComment() メソッドを呼び出している．また変数 `FLAG_COMMENT` は static, final 修飾子を用いて定義されており，定数として用いられている．本来ならば，このメソッド全体を親クラスに引き上げるのであるが，このように各子クラスのフィールド変数をメソッド getCommentCommand() で用いているので，このケースでは 2 つの RVK 変数を引数

```
private void getCommentCommand(Commandline cmd) {
    if (getComment() != null) {
        /* Had to make two separate commands here because if a space is
           inserted between the flag and the value, it is treated as a
           Windows filename with a space and it is enclosed in double
           quotes ("). This breaks clearcase.
        */
        cmd.createArgument().setValue(FLAG_COMMENT);
        cmd.createArgument().setValue(getComment());
    }
}
```

(a) コードクローンのソースコード



(b) クラス階層におけるコードクローンの位置関係

図 22: 「Pull Up Method」を適用したコードクローンの例（適用前）

として受け取るメソッドを ClearCase クラスに作成し，メソッド getCommentCommand() の内部で親クラスに定義したメソッドを呼び出すように変更した（図 23 参照）.

一方，図 24 は「メソッドの抽出」の条件を用いて抽出した $POP(C)$ 最大のコードクローン（=7）を表している．この if 文単位のコードクローンでは，それぞれ GUI の部品である Button クラスのインスタンスを生成している．これらのコードクローン間では対象となるインスタンス変数が異なるのみであり，全く同様の操作を行なっている．このクローンクラスに対して，図 24 に示すように，コードクローン内で用いられている Button 型のインスタンス変数と，2 つの String 型定数を引数として,「メソッドの抽出」を適用した．また，抽

```
private void getCommentCommand(Commandline cmd) {
        super.getCommentCommand(cmd,FLAG_COMMENT,getComment());
}
```

(a) クローンになっていたメソッド

```
void getCommentCommand(Commandline cmd, String arg_0, String arg_1) {
        if (arg_1 != null) {
                /* Had to make two separate commands here because if a space is
                   inserted between the flag and the value, it is treated as a
                   Windows filename with a space and it is enclosed in double
                   quotes ("). This breaks clearcase.
                */
                cmd.createArgument().setValue(arg_0);
                cmd.createArgument().setValue(arg_1);
        }
}
```

(b) ClearCase クラスに作成したメソッド

図 23: 「Pull Up Method」を適用したコードクローンの例（適用後）

出したメソッド内でインスタンス化したオブジェクトをメソッド呼び出し元に反映させるために，メソッドには return 文を追加した．抽出したメソッドの呼び出し文ではメソッド内でインスタンス化されたオブジェクトを得るために Button 型のインスタンス変数を用いてメソッドの返り値を受け取っている．

結果として,「メソッドの引き上げ」「メソッドの抽出」の条件を適応して絞り込んだ 15 のクローンクラスのうちの 12 個について集約することに成功した．残りの 3 つのクローンクラスについては 5.1 節で除去できなかった場合と同様の状況であり，取り除くことができなかった．

5.4 ANTLR

次に ANTLR[3] に対しての行なった適用実験について述べる．ANTLR（ANother Tool for Language Recognition）は構文解析器を作成するためのツールである．作成した構文解析器は Java もしくは C++で出力される．ANTLR のソースファイルは 188 個であり，総行数は 42000 行である．

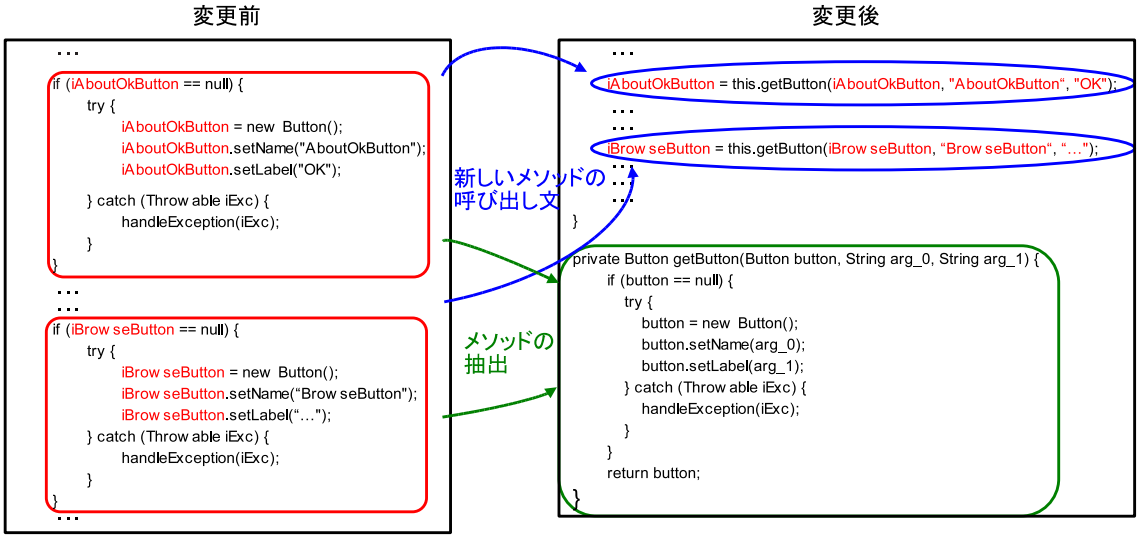
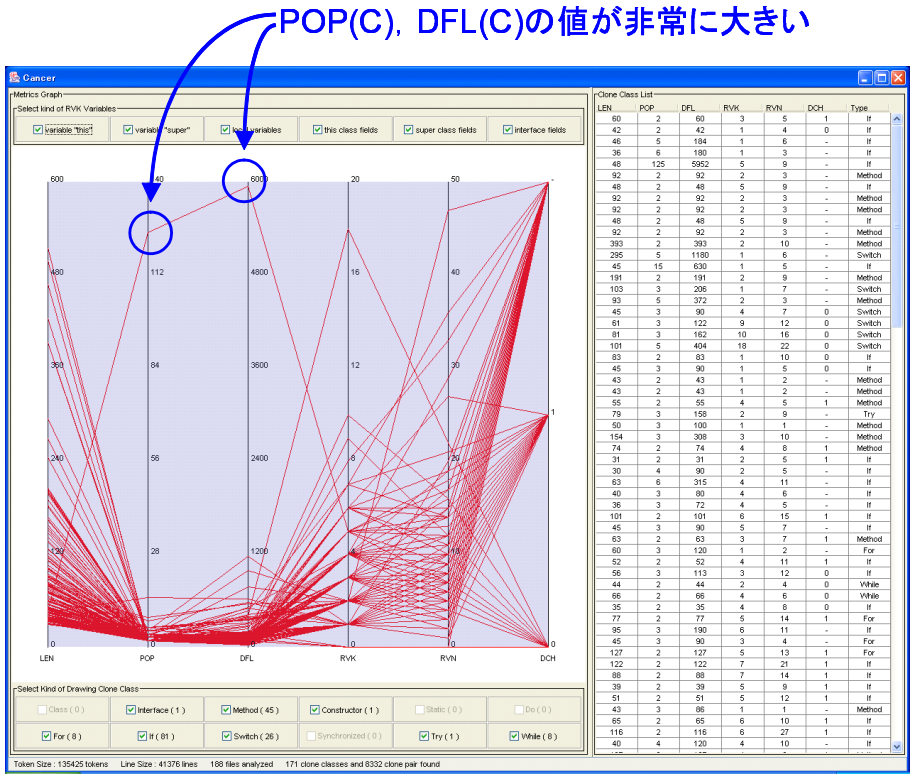


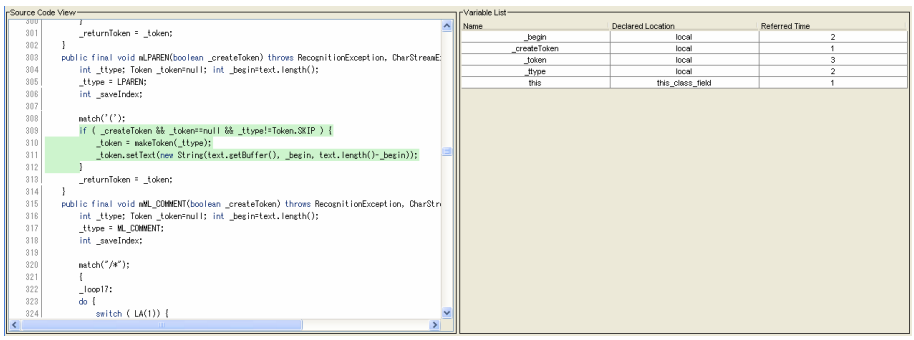
図 24: 「Extract Method」を適用したコードクローンの例

5.4.1 実験方法

解析を行なった結果，171 個のクローンクラスを検出した．図 25(a) は ANTLR に対しての Cancer の実行画面のスナップショットである．この図からみてとれるように $POP(C)$, $DCH(C)$ の値が他のコードクローンに比べ非常に大きいコードクローンが 1 つ存在した．このクローンクラスの $POP(C)$ の値は 125 であり，図 25(b) に示すように if 文単位のコードクローンであった．このコードクローンは $POP(C)$ の値が非常に大きいことから，ANTLR の保守性を著しく低下させていることが予測される．例えば，このコードクローン内で呼び出しているメソッドのシグニチャが変更された場合，ユーザは少なくとも 125 箇所の修正を行なわなければならない．また，このコードクローンは共通の親を持たない 5 クラスに分散して存在した．$RVK(C)$ の値は 5 であり，周囲との結合度がそれほど低いということもなく，本手法で提案したメトリクスを用いた絞り込みを行なった場合は，このコードクローンはリファクタリング対象とはならない．しかし，このケースでは他のクローンクラスに比べ非常に $POP(C)$ の値が大きいことから，このコードクローンを除去すべきものと判断することができ，リファクタリングを行った．このクローンクラスは 5 つのクラスに分散していたことから，それぞれのクラスにおいて「メソッドの抽出を行なった」また，このクローンクラスの除去を行なった後，ANTLR の振る舞いがリファクタリング前と変わっていないことを保証するために回帰テスト行なった．回帰テストには ANTLR 付属の構文定義ファイル 84 個全てを用いた．この定義ファイルは ANTLR に入力として与えるためのものである．



(a) 絞り込みの様子



(b) コードクローンのソースコード

図 25: ANTLR

ANTLR は入力を受け取ると，Java もしくは C++でその定義ファイルの構文解析器を出力する．リファクタリングを行なった ANTLR と行なっていない ANTLR に対して同様に入

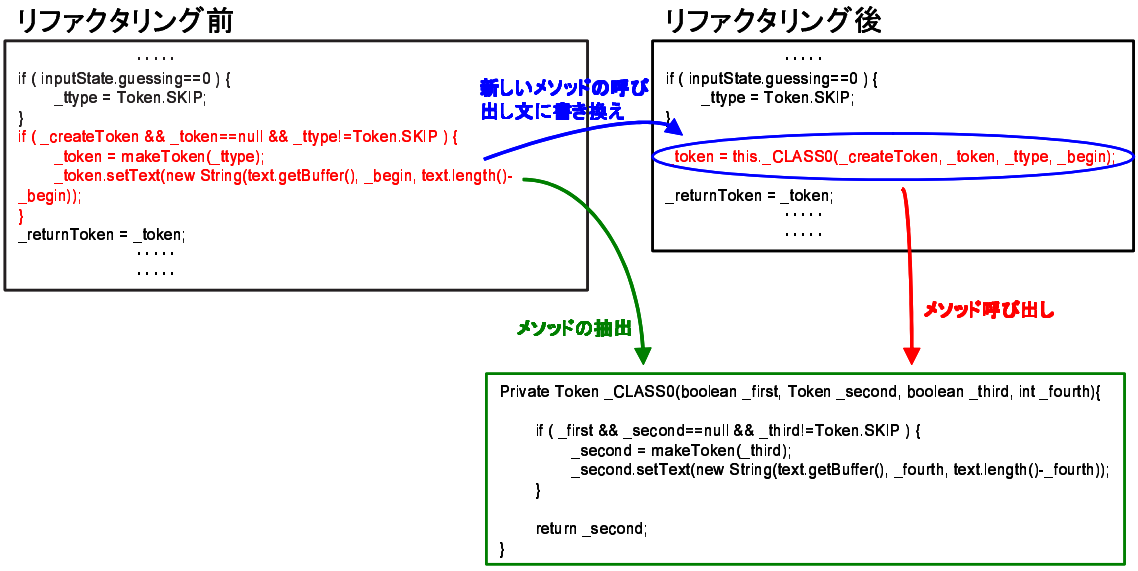


図 26: リファクタリングの様子

力を与え，2 つの ANTLR が出力したソースコードを UNIX コマンドの diff を用いることにより比較した．結果として，2 つの ANTLR から出力されたソースコードは 84 個すべての定義ファイルについて，全く同一であった．

5.4.2 リファクタリング

5.4.1 節で示したようにこのコードクローンは 5 つのクラスに分散して存在した．また，このクローンクラスの $DCH(C)$ の値は-1 であり，共通の親クラスを持っていなかった．このことから，このコードクローン全てを 1 つに集約することは困難であり，クラス毎に「メソッドの抽出」を試みる．図 26 はこのリファクタリングの様子を表している．リファクタリング前，このコードクローン内では外部定義の変数が 5 つ用いられていた．その内 1 つは自クラスのフィールド変数であり，このリファクタリングは 1 つのクラスで閉じているので，このフィールド変数は抽出するメソッドの引数にする必要はない．一方，残りの 4 つはローカル変数であり，抽出するメソッドに引数として与える必要がある．コードクローンになっている if 文の処理内容に注目すると，この if 文の内部では，変数`_token` に新たにオブジェクトを割り当てているのがわかる．つまり，4 つのローカル変数の内，このコードクローン内で差すメモリ領域が変わっている変数は`_token` であり，この変更をメソッドの呼び出し元に反映させる必要がある．よって抽出するメソッドの返り値は`_token` を引数として置き換えた`_second` となる．

6 むすび

本研究では，中規模〜大規模ソフトウェアに対しても実用的な時間で適用可能な，リファクタリングを対象としたコードクローンの抽出，集約手法を提案した．コードクローン抽出には，高速なコードクローン検出ツール CCFinder を用いることにより，大規模なソフトウェアから実用的な時間でリファクタリングの適用単位となるコードクローンの抽出を行なう手法を提案した．また，抽出したコードクローンの特徴をメトリクスとして数値化することにより，そのコードクローンの集約方法の補助を試みた．そして提案手法を実装した，リファクタリング支援ツール Cancer を試作した．

また実際にツールを用いて適用実験を行なった．適用実験では，中規模のソフトウェアに対しても数分で解析可能であり，抽出したコードクローンの多くは集約可能であり，本手法の有効性を示すことができた．しかし，解析不足による適合率の低下は否めず，効率的な集約作業を阻害していることが予想される．

最後に今後の課題としては，まず，適合率を向上させるための解析手法，メトリクスの提案があげられる．また対象とするリファクタリングパターンの増加も考えられる．現在では主に 2 つのリファクタリングパターンのみしか対象となっていない．例えば，クラス単位での類似度をはかり，ある程度の類似度がある場合は，コードクローンとなっている部分を類似クラスの親クラスとして定義することが考えられる．また，現在は，コードクローンを「集約できる」,「集約できない」に基づいて解析を行なっており，ソフトウェアの品質，複雑度などの側面から見た場合の「集約すべき」「集約すべきでない」は考慮されていない．今後は解析対象にこれらのことを含め，より総合的なリファクタリング支援を行なうことのできる解析手法の提案を行なうことを考えている．

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻　井上克郎教授に心より深く感謝致します.

本論文を作成するにあたり，常に適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻　楠本真二助教授に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻　松下　誠助手に心から感謝致します.

本研究を通して，逐次適切な御指導，御助言を頂きました 独立行政法人 科学技術振興機構 さきがけ「機能と構成」領域研究員 神谷年洋 氏に深く感謝致します.

本研究において，様々な御協力を頂きました 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 高尾 祐治 氏に深く感謝します.

本研究において，様々な御協力を頂きました 大阪大学 基礎工学部情報科学科中山 崇 氏に深く感謝します.

最後に，その他様々な御指導，御助言等を頂いた 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 井上研究室の皆様に深く感謝いたします.

参考文献

[1] A. Aiken, "A System for Detecting Software Plagiarism (Moss Homepage)", `http://www.cs.berkeley.edu/~aiken/moss.html` [Last visited 1st Feb. 2003].

[2] Ant, `http://ant.apache.org`, 2003.

[3] ANTLR, `http://www.antlr.org`, 2003.

[4] B. S. Baker, *A Program for Identifying Duplicated Code*, Computing Science and Statistics, 24:49-57, 1992.

[5] B. S. Baker, *On Finding Duplication and Near-Duplication in Large Software Systems*, IN Proc. IEEE Working Conf. on Reverse Engineering, pages 86-95, July 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] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring", *Proceedings the 7th Working Conference on Reverse Engineering*, 2000, 98-107.

[8] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Measuring Clone Based Reengineering Opportunities", *Proceedings 6th IEEE International Symposium on Software Metrics*, 1999, 292-303.

[9] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Partial redesign of Java software systems based on clone analysis", *Proceedings 6th IEEE International Working Conference on Reverse Engineering*, 1999, 326-336.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, *Clone Detection Using Abstract Syntax Trees*, Proc. IEEE Int'l Conf. on Software Maintenance (ICSM) '98, pages 368-377, Bethesda, Maryland, Nov. 1998.

[11] E. Burd, and J. Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance", *Proceedings 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, 2002, 36-43.

[12] M. Dorfman, and R. H. Thayer, *Software Engineering*, IEEE Computer Society Press, 1997.

[13] S. Ducasse, M. Rieger, and S. Demeyer, *A Language Independent Approach for Detecting Duplicated Code*, Proc. of IEEE Int'l Conf. on Software Maintenance(ICSM) '99, pages 109-118, Oxford, England, Aug. 1999.

[14] N. Ford, and M. Woodroffe, *Introducing software engineering*, Prentice-Hall, 1994.

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

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

[17] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue, *On software maintenance process improvement based on code clone analysis*, Proc. of Profes 2002, pp. 185-197 (2002).

[18] Y. Higo, T. Kamiya , S. Kusumoto , K. Inoue, *On Refactoring for Open Source Java Program*, The 9th IEEE International Software Metrics Symposium (Metrics 2003), pp.247-251 ,Sydney , Australia , (September 3-5 , 2003).

[19] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, *Refactoring Support Based on Code Clone Analysis(to appear)*, The 5th International Conference on Product Focused Software Process Improvement(Profes 2004), Kyoto-Nara, Japan , (April 5-8 , 2004).

[20] 肥後 芳樹, 神谷 年洋, 楠本 真二, 井上 克郎, “類似コード片を利用したリファクタリングの試み”, 情報処理学会研究報告 Vol.2003 , No.73 , pp.29-36 , (2003 / 7 / 17)

[21] Y. Higo, Y. Ueda , T. Kamiya , S. Kusumoto, K. Inoue , *Extension of Code Clone Analysis System for Refactoring Activities*, 電気関係学会関西支部連合大会 (2002 / 11 / 9)

[22] *IEEE Std 1219: Standard for Software Maintenance*, 1997.

[23] 井上克郎, 神谷年洋, 楠本真二, “コードクローン検出法”, コンピュータソフトウェア, vol.18, no.5, pp.47-54, 2001.

[24] JavaCC, `http://javacc.dev.java.net`, 2003.

[25] J. Ferrante, K.J. Ottenstein, J.D. Warren, *The Program Dependence Graph and Its Use in Optimization* ACM Transactions on Programming Languages and Systems, vol. 9, Issue. 3, pp. 319-349, (1987-7).

[26] JUnit, `http://www.junit.org`, 2003.

[27] J. H. Johnson, *Identifying Redundancy in Source Code using Fingerprints*, Proc. of CASCON ’93, pages 171-183, Toronto,Ontario, 1993.

[28] T. Kamiya, S. Kusumoto, and K. Inoue, *CCFinder: A multi-linguistic token-based code clone detection system for large scale source code* IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654-670, (2002-7).

[29] 加藤 博己, “データベースのビジュアルな検索と分析 (OLAP)”, IPSJ Magazine Vol.41 No.4 pp. 363 - 368, 2000.

[30] R. Komondoor and S. Horwitz, *Using slicing to identify duplication in source code*, In Proc. of the 8th International Symposium on Static Analysis, Paris, France, July 16-18, 2001.

[31] Jens Krinke, *Identifying Similar Code with Program Dependence Graphs* , In Proc. of the 8th Working Conference on Reverse Engineering, 2001.

[32] J. Mayland, C. Leblanc, and E. M. Merlo *Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics*, Proc. of IEEE Int’l Conf. on Software Maintenance (ICSM) ’96, pages 244-253, Monterey, California, Nov. 1996.

[33] L. Prechelt, G. Malpohl, M. Philippsen, *Finding plagiarisms among a set of programs with JPlag*, submitted to Journal of Universal Computer Science, Nov. 2001, taken from `http://wwwipd.ira.uka.de/~prechelt/Biblio/`

[34] M. Rieger, S. Ducasse, *Visual Detection of Duplicated Code*, 1998.

[35] Pigoski T. M, *Maintenance*, Encyclopedia of Software Engineering, 1, John Wiley & Sons, 1994.

[36] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *Gemini: Maintenance Support Environment Based on Code Clone Analysis*, 8th International Symposium on Software Metrics, June 4-7, 2002.

[37] 植田泰士, 神谷年洋, 楠本真二, 井上克郎 “開発保守支援を目指したコードクローン分析環境”, 電子情報通信学会論文誌 D-I, Vol.86-D-I, No.12, pp.863-871, June. 2003.

[38] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *On Detection of Gapped Code Clones using Gap Locations*, Submitted to 9th Asia-Pacific Software Engineering Conference, 2002.

[39] S. W. L. Yip and T. Lam, *A software maintenance survey*, Proc. of APSEC ’94, pages 70-79, 1994.