修士学位論文

題目

開発保守支援を目指したコードクローンの抽出手法
- 支援環境 Gemini への実装 -

指導教官
井上克郎 教授

報告者
植田泰士

平成 15 年 2 月 12 日

大阪大学 大学院基礎工学研究科
情報数理系専攻 ソフトウェア科学分野

平成 14 年度 修士学位論文

開発保守支援を目指したコードクローンの抽出手法
- 支援環境 Gemini への実装 -

植田泰士

内容梗概

ソフトウェア保守を困難にする要因の一つとしてコードクローンが指摘されている．コードクローンとは，ソースコード中に含まれる同一または類似したコード片のことである．それらは多くの場合，既存システムに対する変更や拡張時における，コピーとペーストによる安易な機能的再利用の際に発生する．

我々は，コードクローンを全て検出するツール CCFinder，およびその解析結果の参照支援システムを開発し，これまで様々なソフトウェアに対して適用を行った．それらの適用により，実用的な立場から，CCFinder とその参照支援システムには，大規模ソフトウェアに対して検出されたコードクローン情報の識別性の欠如が問題として取り上げられていた．また同時に，実開発においてコピーとペーストによってコードクローンが生成される際にはペーストコード片に対して部分的な修正が加えられる場合が多いが，CCFinder では部分的な修正が加わったコードクローンの直接的な検出ができないことも問題となっていた．

本研究では，コードクローン情報を利用した開発保守支援を目指し，不一致部分を含んだコードクローンの検出手法の提案，および検出されたコードクローンの識別性を向上するため，実利用を目的としたコードクローンの抽出手法の提案を行う．また本手法を，支援環境 Gemini に実装し，有効性を確認するため幾つかのソフトウェアに適用実験を行った．適用実験により，本手法を用いることで，CCFinder のみでは発見が困難であった部分的な修正が加わったコードクローンを容易に発見することができ，また実利用として特にリファクタリングを行えるようなコードクローンを容易に抽出可能であることを確認した．

主な用語

コードクローン
ソフトウェア保守
ソフトウェアメトリクス

目 次

1 まえがき **4**

2 ソフトウェア保守とコードクローン **6**

2.1 ソフトウェア保守 6

2.1.1 ソフトウェア保守と保守における問題点 6

2.1.2 リファクタリング 7

2.2 コードクローンと既存のコードクローン検出法 10

2.3 コードクローン検出ツール CCFinder 13

2.3.1 概要 13

2.3.2 コードクローン検出処理手順 15

2.3.3 検出例 15

3 開発保守支援を目指したコードクローンの抽出手法 **17**

3.1 これまでの適用における CCFinder に対する評価 17

3.2 不一致を含んだコードクローンの検出 19

3.2.1 コードクローンの分類 19

3.2.2 Gapped クローンの検出の必要性 20

3.2.3 Gapped クローンの検出 20

3.3 コードクローン情報の利用とそれに求められる抽出 30

3.3.1 利用目的 30

3.3.2 利用に求められる分析・抽出 31

3.3.3 機能的なまとまりを持ったコードクローンの抽出 33

3.3.4 メトリクスを用いたコードクローンの抽出 34

4 コードクローン分析支援環境 Gemini **40**

4.1 コードクローン情報利用プロセス 40

4.2 システムの構成 42

4.3 実装 43

5 評価 **45**

5.1 Gapped クローンの検出の適用実験 45

5.1.1 概要 45

5.1.2 分析 46

5.2 機能的なまとまりを持ったコードクローンの抽出の適用実験 50

5.2.1 概要 50
5.2.2 分析 50
5.3 メトリクスを用いたコードクローンの抽出の適用実験 54
5.3.1 概要 54
5.3.2 分析 54

6 むすび **71**

謝辞 **72**

参考文献 **73**

付録 **77**

1　まえがき

近年，ソフトウェアシステムの大規模化，複雑化に伴い，プログラムの保守・デバッグ作業に要するコストが増加してきている．ソフトウェア保守を困難にしている一つの要因としてコードクローンが指摘されている．

コードクローンとは，ソースコード中に含まれる同一または類似したコード片のことである．それらは多くの場合，既存システムに対する変更や拡張時における「コピーとペースト」による安易な機能的再利用の際に発生する．しかし，もしあるコード片にフォールトが含まれていた場合，そのコード片に対する全てのコードクローンについて修正を行わなければならない．また，保守性を高くするため，同値類を一つのサブルーチン等にまとめるのがよい場合もあるかもしれない．そのためには，コードクローンを全て検出することが必要となる．

これまで様々なコードクローン検出法が提案されている．我々の研究グループもトークン単位でのコードクローンを検出するツール (CCFinder[30]) を開発してきており，これまで様々なソフトウェアに対する適用を行った．これらの適用の中で，CCFinder は大規模なソースコードも，細粒度でのコードクローン解析を非常に高速に行うことが可能であると評価されてきた．

しかし，CCFinder から検出されてくるコードクローン情報は，ソフトウェア規模が大きくなればなるほど，当然膨大なものとなる．ソースコード中から手作業で類似部分をもれなく探す手間がなくなったとはいえ，現実的に，これらの膨大なコードクローン情報は，それぞれが何らかの有効な情報であるかどうかをひとつひとつ目視チェックしていくことは不可能である．そこで，我々はさらに CCFinder の解析結果の参照支援システムを試作し [42][43][44] ，コードクローンの位置情報の視覚化と，コードクローンに対する単純な評価尺度を用いてソースコードの参照支援を試みた．そのシステムにおいては，規模の小さなソフトウェアであれば，視覚化されたコードクローンの位置関係や，そのコード片の長さや数等で着目すべきコードクローンが容易に判別可能であった．しかし，大規模ソフトウェアともなると，個々の特徴は膨大な情報の中に埋没し，実際の開発保守現場での利用は困難であった．こういった実用的な立場から，CCFinder とその参照支援システムには大規模ソフトウェアに対するコードクローン情報の識別性，有意性が欠如していることが問題として取り上げられていた．当然，識別性，有意性を確保していくためには，なんらかの利用目的の策定とその特徴を捉えた抽出が必要となる．

また実際「コピーとペースト」によってコードクローンが作りこまれる場合，ペーストしたコード片は何も変更が行われないことはまれである．こういった際，CCFinder は，識別子名等の違いまでしか吸収できないため，例えば新規処理コードの追加があった箇所等は不

一致部分として判断され，複数の相対的に短いクローンペアに分割されて検出される．短いコードクローンには興味がなく，もしある長さ以上のコードクローンだけしか検出しなかったとしたら不一致部分により分割され短くなると検出されないことになる．逆に，短いものまで検出するようにすると全体で検出されるコードクローン数が膨大になり，その他の情報の中に埋もれてしまう．つまり，フォールトが含まれている可能性のあるコード片も見落としてしまいかねない．よって，この不一致部分を含んだコードクローンをいかに識別・検出するかについても課題となっていた．

そこで，本研究においては，コードクローン情報を利用した開発保守支援を目指し，不一致部分を含んだコードクローンの検出手法の提案，および実利用を視野にいれたコードクローンの抽出手法について提案を行う．また，本手法を，支援環境 Gemini に実装し，いくつかのソフトウェアを対象にして適用実験を行い，本手法の有効性を確認する．

以降，2 節では，ソフトウェア保守における問題やコードクローン検出に関連した研究について述べ，3 節では，これまでの適用における CCFinder に対する評価および，それらから必要であると分かった不一致部分を含んだコードクローンの検出やコードクローンの抽出手法について提案を行う．また，4 節では，これらの手法を実装した支援環境 Gemini に関して説明を行う．5 節では，いくつかのソフトウェアを対象にして適用実験を行い，その結果についての分析と考察を行う．最後に 6 で，本研究のまとめと今後の課題について述べる．

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 [26]．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．

また，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 [21]，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[16] 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，公開したくないビジネス中核を含まないソフトウェアについては，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 [41] 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 [20]．

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．また特に，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 [20]．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コールに置き換える．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，それらをスーパー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．

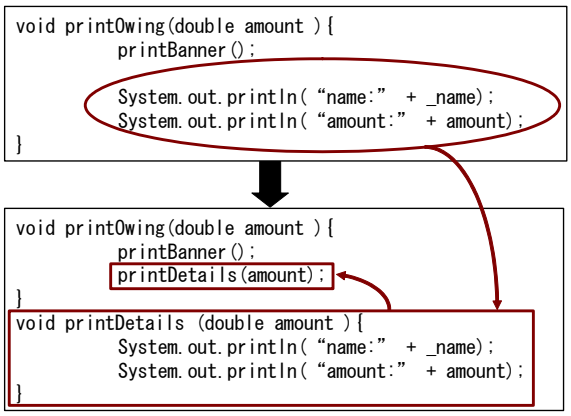


図 1: リファクタリング (メソッドの抽出)

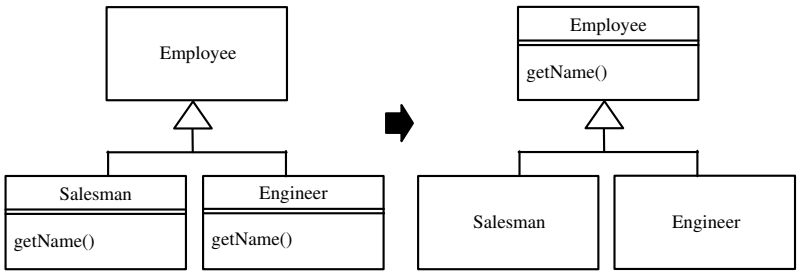


図 2: リファクタリング (メソッドの引き上げ)

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，スーパークラスを作成して，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 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，元のメソッドが同一になるように，各処理を同じシグニチャを持つメソッドに，スーパークラスに引き上

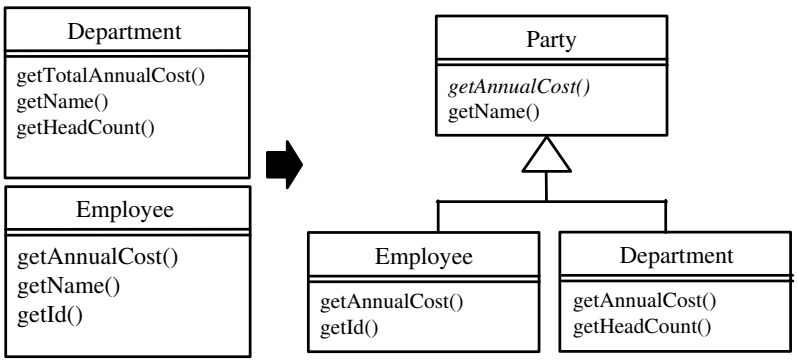


図 3: リファクタリング (スーパークラスの抽出)

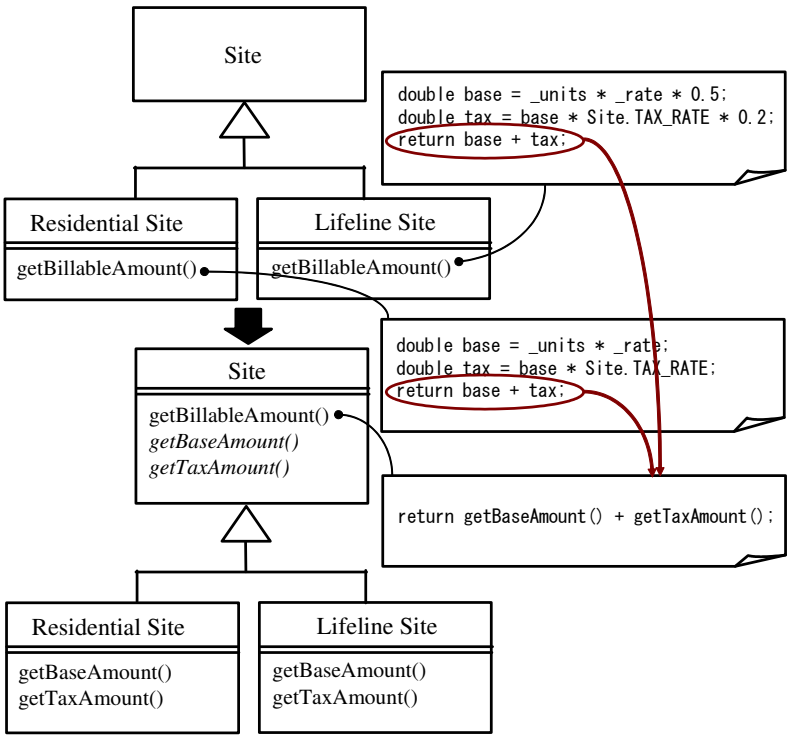


図 4: リファクタリング (テンプレートメソッドの形成)

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，順序の制御をスーパークラスに移動し，異なる処理については，異なったまま行わせる (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

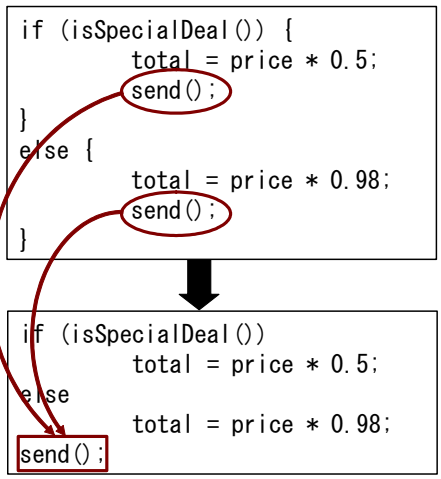


図 5: リファクタリング (重複した条件記述のコード片の統合)

件記述以外にも，例外記述にも適用可能である．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，finallly ブロックに移動する．

ポリモーフィズムを用いたスイッチ文の削除

スイッチ文などは重複したコードを生成しやすくしている．同じようなスイッチ文がある場合は，新たな分岐を追加した際に全てのスイッチ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というものが必要な作業であり，リファクタリングを行うことで，よりコードの理解が深まり，バグを混入しにくくなり，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．

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 [12][30] ．

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．

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っている．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 [27]．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，現実的に困難である．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][6][7][8][9][10][11][12][17][30][33][34][36][40] ．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 は次節 2.3 参照)．

Covet

文献 [36] 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[12]

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][8]

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 [22] を用いているため線形時間で解析可能である．

Duploc[17]

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[40]

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 [33]

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．[33] 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 [34]

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[9][10][11]

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 [13] 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 概要

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 (図 6 参照) 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 (図 6 参照) 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．

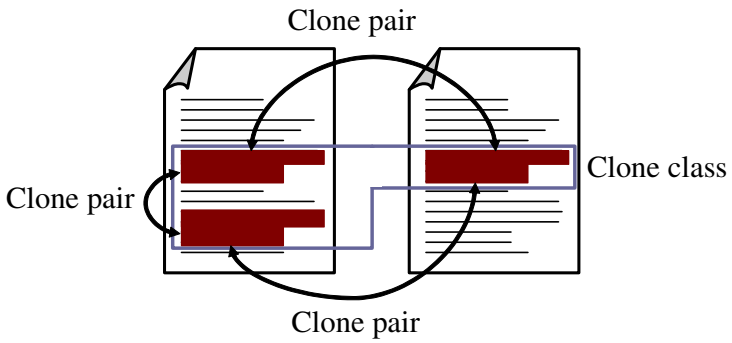


図 6: クローンペアとクローンクラス

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 [27]．

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．

繰り返しコードに対して特別な処理を行う
(3.3.4 節，$LNR(C)$ の定義参照)

2.3.2 コードクローン検出処理手順

CCFinder のコードクローン検出処理は，以下の 4 ステップで構成されている．

ステップ 1: 字句解析: ソースコードをプログラミング言語の文法に沿ってトークン列に変換する．その際，空白とコメントは機能に影響しないので無視される．ファイルが複数の場合には，単一ファイルの解析と同じように処理できるよう，単一のトークン列に連結する．

ステップ 2: 変換処理: 実用的に意味を持たないコードクローンを取り除くため，もしくはある程度の違いは吸収するためにトークン列を実用的に意味のあるコードクローンのみを検出するための変換を施す．例えば，変数名，関数名などは全て同一のユニークなトークンに置換される．

ステップ 3: 検出処理: トークン列を比較して，一致した部分トークン列をコードクローンとする．但し，指定された最小一致トークン以上の長さをもつコードクローンのみが検出される．また，トークン列の比較にはサフィックス木探索 [22] を行うため，線形時間[1]で解析可能となる．

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．

2.3.3 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 には，Java で互いに似通った 2 つのメソッドが書かれ，図中の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

[1]構築されたサフィックス木からコードクローンを取り出すには，木の全探索が必要であるため，トークン列の長さを n，(極大コードクローンだけではなく全ての) クローンペアの数を k をとして $O(n+k)$.

```
    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

目-10 行目) と B2 (20 行目-22 行目), そして C1 (12 行目) と C2 (25 行目) がそれぞれクローンペアとして検出される．それぞれのクローンペアの長さは順に 7, 18, 6 トークンとなっている．見ての通り，A1 と A2 の間，B1 と B2 の間には次のような幾らかの違いが含まれているがコードクローンとして検出可能となっている．

- 名前空間の違い (e.g. “org.apache.regexp.RE” と “RE”).
- 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，2.3.1 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．

3　開発保守支援を目指したコードクローンの抽出手法

3.1　これまでの適用における CCFinder に対する評価

これまで，我々は CCFinder を様々なソフトウェアに対して適用を行ってきた．フリーソフトウェアとしては，JDK のライブラリ群, Linux, FreeBSD, OpenBSD，NetBSD 等の数 MLOC 規模の大規模なオープンソースソフトウェアに，商用ソフトウェアとしては，ソフトウェア開発を行っている現場の企業約 10 社程度において実験的な適用を試みてきた．その他にも，ソフトウェア著作権訴訟においては類似ソフトウェア間における CCFinder を用いた比較結果が証拠として提出され，教育現場においては大学のプログラミング演習の講義で学生が作成したソフトウェアへの適用が行われた．これらの適用の中で，CCFinder は大規模なソースコードも，細粒度でのコードクローン解析を非常に高速に行うことが可能であると評価されてきた．

しかし，CCFinder から検出されてくるコードクローン情報は，ソフトウェア規模が大きくなればなるほど，当然膨大なものとなる．例えば，JDK ライブラリ群，約 1.2MLOC の中には，30 トークン以上のクローンペアが約 32 万組検出される．CCFinder によりソースコード中から手で類似部分をもれなく探す手間がなくなったとはいえ，現実的に，これらの膨大なコードクローン情報は，それぞれが何らかの有効な情報であるかどうかをひとつひとつ目視チェックしていくことは不可能である．それは，それぞれのコードクローンの実際のソースコード内容を参照する支援系のあるなしに関わらず，数が膨大すぎるが故に調べ尽くすコストが現実的でないことに変わりはない．

経験的にも，CCFinder から検出されたそのままの情報のみでは，大規模ソフトウェアにおいては，ソフトウェア管理者等が，対象ソフトウェアの中にどの程度類似しているものが存在するのかを確認する，もしくはどのあたりのモジュールにコードクローンが集中して存在しているのかを知る程度の利用に留まざるを得ない．事実，我々は CCFinder の解析結果の参照支援システムを試作し [42][43][44]，コードクローンの位置情報の視覚化と，コードクローンに対する単純な評価尺度を用いてソースコードの参照支援を試みた．そのシステムにおいては，規模の小さなソフトウェアであれば，視覚化されたコードクローンの位置関係や，そのコード片の長さや数等で着目すべきコードクローンが容易に判別可能であった．しかし，大規模ソフトウェアともなると，個々の特徴は膨大な情報の中に埋没し，実際の開発保守現場での利用は困難であった．

こういった実用的な立場から，一般にコードクローン検出ツールには次のような特性が求められている [27]．

特性 1 大規模プログラムへの適用可能性:

大規模なソフトウェアに対しても現実的な時間とメモリの範囲で検出できる．

特性 **2** コードクローンの識別性:

何らかの特徴により，着目すべきコードクローンを識別することができる．

特性 **3** コードクローンの有意性:

現実的に意味を持たないコードクローンを取り除くことができる．

特性 **4** 拡張性:

コードクローン検出対象プログラムの言語に深く依存しない．

2.3 節から CCFinder は，特性 1 および特性 4 は十分に満たしていると言える．しかし，Burd らは，評価実験 (対象言語は Java) において，CCFinder の検出するコードクローンの健全性は 72%であったとしている [13]．コードクローンの定義自体は，2.2 節で述べたようにツールの検出アルゴリズム毎に定義されるものであり，現実的に意味のあるコードクローンであるかどうかを判断するのは主観的な評価にならざるを得ないが，某 A 社からの CCFinder 試行報告書 (対象言語は COBOL) でも，前後に不要なステップが含まれていたりと，意味のないコードクローンも多く検出されたと報告があった．つまりは，大規模ソフトウェアに対するコードクローン情報の識別性，有意性の欠如が問題として取り上げられていた．これらの膨大な情報に有効な識別性，有意性を確保していくためには，当然，なんらかの利用目的の策定とその特徴を捉えた抽出が必要となる．

また，2.2 節で述べたように，コードクローンは，既存システムに対する変更や拡張時における「コピーとペースト」によるプログラミングにより作りこまれることが多い．ペーストしたコード片はそのまま利用されるとは限らない．実際，ペースト先のプログラムコンテキストに合わせた部分的な修正，変更が行われる可能性は高い．それは，変数名，関数名等の識別子名の変更だけではなく，ペーストされたコード片の間への新規処理コードの追加，ペーストコードの部分的な削除，変更も起こりうる．むしろ，何も変更が行われない場合の方がまれである．こういった際，CCFinder は，識別子名等の違いまでしか吸収できないため (2.3 節参照)，例えば新規処理コードの追加があった箇所等は不一致部分として判断され，複数の相対的に短いクローンペアに分割されて検出される．短いコードクローンには興味がなく，もしある長さ以上のコードクローンだけしか検出しなかったとしたら不一致部分により分割され短くなると検出されないことになる．逆に，短いものまで検出するようにすると全体で検出されるコードクローン数が膨大になり，その他の情報の中に埋もれてしまう．よって，この不一致部分を含んだコードクローンをいかに識別・検出するかについても課題となっていた．

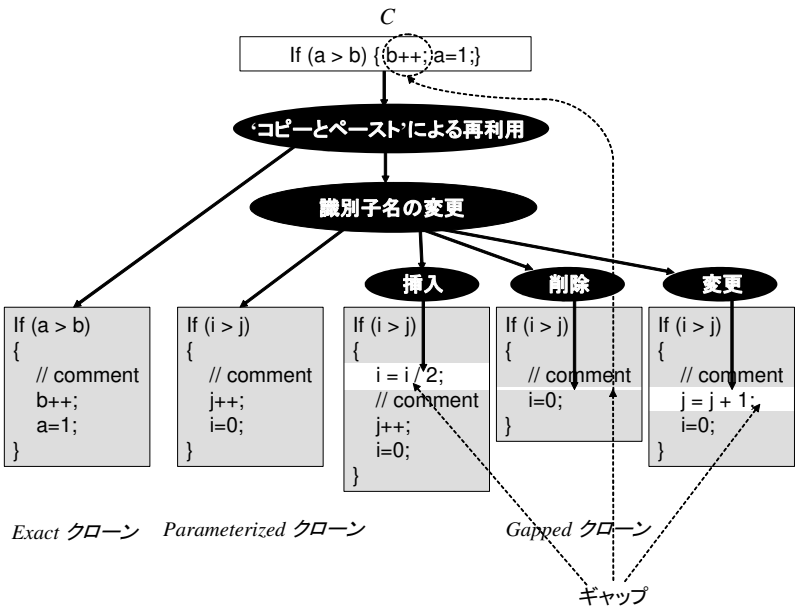


図 8: コードクローンの分類

次節でまず不一致部分を含んだコードクローンの検出について述べた後，3.3.1 節以降開発保守支援を目指したコードクローン情報の利用目的，その目的に即した抽出について述べる

3.2 不一致を含んだコードクローンの検出

本節では，不一致部分を含むコードクローンを検出するが，まず，それ以外のコードクローンとの定義の違いを明確にし，そしてその不一致を含んだコードクローンを検出する必要性について説明する，その後，我々の提案する検出手法について述べる．

3.2.1 コードクローンの分類

ここで我々は，コード片 C に対するコードクローンを次の 3 種類に分類する．

Exact クローン

C に対し，プログラムテキストとして完全に一致しているコード片．但し，文字列レベルでの一致を指すものではなく，空白，改行，コメントなどの違いは考慮しない．

Parameterized クローン

C に対し，変数名，定数名，クラス名，メソッド名等のユーザ定義名の違いを除き，一致している．つまり言語依存の予約語等が一致し，構文的に同じであるコード片．

Gapped クローン

C に対し部分的に類似しているコード片．つまり，構文的にも一致しない不一致コード (ギャップ (gap)) を，部分的に含む．

我々の興味あるギャップとは，コピーとペーストによるプログラミングにおいて，ペーストされたコード片に含まれる，新規追加されたコード，削除されたコード，または構文が変更されたようなコードである．図 8 に，Exact クローン，Parameterized クローン，Gapped クローンの例を示す．定義より，CCFinder が検出可能なコードクローンは，Exact クローンおよび Parameterized クローンとなる．以降，この，Exact クローンおよび Parameterized クローンを併せて，**Ng クローン (Non-gapped clone)** と呼ぶ．

3.2.2 Gapped クローンの検出の必要性

2.3.3 から，図 7 中の 2 つのメソッドは，かなり類似していたことが分かる．

しかし，2.3.1 節で述べたように，CCFinder では，コードクローン検出の際，検出するコードクローンの最小一致長が定められる．2.3.3 節の検出例においては，最小一致トークン数を 5 トークンに定め，3 つのクローンペア (A，B，C) しか検出されなかったが，一般的には，最小一致トークン数を 5 トークンという比較的小さな値に定めれば，非常に多くのコードクローンが検出されてくる．

逆に最小一致トークン数を大きくして，15 トークンに定めると，B マークが付されたコード片のみが，クローンとして検出され，A，C マークが付されたコード片は，検出されなくなる．さらに最小一致トークン数を 20 トークンに大きくすれば，コードクローンは全く検出されない．

そこで，ある一定の長さ 10 トークンまでのギャップをコードクローン中に含むことを許してみることにする．そうすると，A,B,C のマークを付されたコード片達が全て結合される．つまり，最小一致トークン数を大きめの 15 トークンや 20 トークンに定めたとしても，その結合されたものはそれ以上の長さを持っているため，ひとつのコードクローンとして検出することができる．明らかに，B だけからから判断するよりも，このひとつの大きなコードクローンから 2 つのメソッド間の類似性を判断する方が容易である．ゆえに Gapped クローンを探すことができれば，より効率よくコードクローンに基づいた分析を行うことができると言える．

3.2.3 Gapped クローンの検出

アプローチ

そもそも，仮に Ng クローンの位置情報が存在した場合，本研究でいう Gapped クローンの

個々の発見問題は，どの Ng クローンの組合せを Gapped クローンにするかという，Ng クローンの組み合わせ決定問題に帰着する．

しかし，もし複数の Ng クローンが互いに重なりあい密集していた場合は，あるひとつの Ng クローンからの次の Ng クローンへの結合先が多数存在する．その場合，全体としての Gapped クローン検出は，組合せ爆発を起こす．実際，そのように互いに Ng クローンが密集していることは多い．そのため，個々の Gapped クローンを全て検出するということは大規模ソフトウェアには向かない．

しかし，実用的な立場から，3.2.2 節のように類似性しているコード位置を認識・識別するには，個々の Gaapped クローンまで知る必要性はなく，密集範囲全体としてどのあたりが類似しているかが識別可能であればよい．

そこで我々は Gapped クローンの検出に際し，実際に個々の Gapped クローンを検出するのではなく，互いに素な Ng クローンの連結集合の検出によるマンマシン共同作業のアプローチをとる．

Gapped クローン検出プロセス

図 9 に Gapped クローン検出プロセスを示す．本プロセスは次の 4 つのステップで構成されている．

ステップ 1: Ng クローン検出，

ステップ 2: ギャップ検出，

ステップ 3: 視覚化，

ステップ 4: ソースコード分析．

ステップ 1 においては，入力されたソースコードから Ng クローンを検出する．次にステップ 2 で，ステップ 1 で検出された Ng クローンの位置情報からギャップの位置情報を生成する．そしてステップ 3 において，それら Ng クローン位置情報とギャップ位置情報を視覚化する．最後にステップ 4 で分析者が，視覚化された情報を利用して，Gapped クローンのソースコードを分析するという手順である．

以下に，例を用いて検出プロセス手順説明を行う．例として，

コード列 X: “ABCDCDEFBCDG”,

コード列 Y: “ABCEFBCDEBCD”

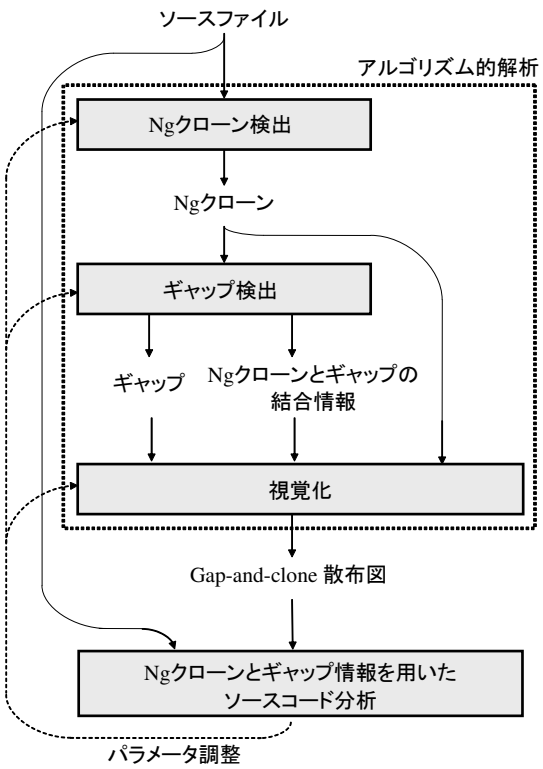


図 9: Gapped クローン検出プロセス

の 2 つのコード列の比較を考える．ここで，“A”,“B”,“C”,. . . の記号は，文字，トークン，行，文，関数などのような，プログラムテキスト中のを表しているものとする．つまり，本節における検出プロセスは，特定の Ng クローン検出アルゴリズムの検出粒度に束縛されることはなく，各記号は，任意の Ng クローン検出ツール毎の検出粒度に対応するものとする．また，我々は複数ファイル間での比較，もしくは単一ファイル内での比較を想定している．上記コード列 X,Y は異なるプログラムテキストであるが，コード列 X,Y が同じ場合も検出プロセスは同様である．

ステップ **1** (Ng クローン検出)

入力ソースコードから **Ng** クローンを検出し，効率的なギャップ検出を行うための準備として **Ng** クローンをソートする．

例における検出結果を，表 1 に示す．本表には，任意の長さの Ng クローンが含まれている (e.g., {c6}:長さ 5, {c1, c2, c3, c5, c7}:長さ 3, {c4}:長さ 2)．

Ng クローンの最小一致長を定めなければ，これら全てが検出されることになる．しかし，この例において，たかだか長さ 12 のコード列の中に 7 つもの Ng クローンが検出

されているように，大規模ソフトウェアになれば，短い Ng クローンは数多くの存在する．代入文，変数宣言のような単純な構文が存在するため，比較的短いクローンの中には単なる偶然の一致が多く，そういったものは実用的に有用でないといえる．したがって，我々の CCFinder において最小一致トークン数が定められるように，一般的に実用性を考慮したコードクローン検出ツールでは，ある閾値より短いコードクローンは検出されないことが多い．よって以降，最小一致長 (以下，閾値 **1**) が指定可能であることを前提とする．

この検出の際に決められる閾値 1 は Gapped クローンの中に含まれる，それぞれ一致部分の最小長となる．つまり，最終的に分析を行いたい Gapped クローンの最小長よりも小さい値が指定されなければならない．

また，ステップ 2 においてギャップを効率よく検出するための前準備として，Ng クローンの検出結果に対してソートをかける．Ng クローンの 2 つのコード片対それぞれは，単一ファイル内で閉じており，複数ファイルをまたぐことがないので，まずファイルの組み合わせで Ng クローン集合を分類することができる．そして，その分類されたそれぞれの集合の中でソートを行う．

ここで，一般性を失うことなく，コード片 X がファイル X，コード片 Y がファイル Y であると考える (ファイル X $\neq$ ファイル Y)．この時，Ng クローン集合に対してファイル X 側のコード片の位置情報でソートをかけ (よりファイルの先頭に近いものを前)，

表 1: Ng クローン検出結果

Ng クローン ID	コード列		一致文字列
	X	Y	
c1	1 - 3	1 - 3	“ABC”
c2	2 - 4	6 - 8	“BCD”
c3	2 - 4	10 - 12	“BCD”
c4	5 - 6	11 - 12	“CD”
c5	5 - 7	7 - 9	“CDE”
c6	7 - 11	4 - 8	“EFBCD”
c7	9 - 11	10 - 12	“BCD”

*(*コード片の位置は記号の開始位置と終了位置を表す．例えば *“6 - 8”* は，*6* 番目の記号から *8* 番目の記号までを意味する．*)*

ファイル X 側のコード片の位置が同じものに関してはファイル Y 側のコード片の位置でソートをかける．同一ファイル内での比較の場合 (ファイル X = ファイル Y) には，2 つのコード片対のうちで，より先頭に近いもの同士でソートかけた後，他方同士でソートをかける．表 1 は，既にソートが完了した結果となっている．

ステップ **2** (ギャップ検出)

ソート済みの **Ng** クローン情報を元に，ギャップの位置情報を生成する．

図 10 にアルゴリズムの詳細を pseudo コードとして示す．各ギャップ毎に生成される情報は，ギャップのソースコード上での位置情報，ギャップの両端に連結される 2 つの Ng クローン情報であり，これらの情報はステップ 3 で利用される．

表 2 は，例において生成されたギャップ情報を示している．これらのギャップ情報 (および Ng クローン) を散布図として表現すれば，図 11(a) のようになる．ここでは，各ギャップをそれぞれ g1, ..., g7 と呼ぶことにする．

図 10 の中で現れている `threshold2`(以降，閾値 **2**) は Gapped クローン中に含まれることが許容されるそれぞれのギャップの上限長を表している．

また，図 10 の中の最適化では，Ng クローン情報がソート済みであるという事実を利用している．内側の `for` ループでは，いったん Ng クローン `ci` からの距離が閾値 2 以上の Ng クローンが見つかれば，それ以降は閾値 2 以内の距離にある Ng クローンが見つかることは有り得ない．ここで注意すべきは，もし閾値 2 が入力ソースコード長に対して十分小さく，全 Ng クローンの密集度合いに偏りがなければ，ある Ng クロー

表 2: ギャップ位置

ギャップ ID	コード列 X	コード列 Y	長さ
g1	4	4 - 6	3
(g2)	4	4 - 10	7
g3	4 - 6	-	3
(g4)	4- 8	4 - 9	6
g5	-	9 - 10	2
(g6)	5 - 8	9	4
g7	8	-	1

```
// 各 Ng クローンについて
for (i = 0; i < ngCloneTotalCount; i++)
{
  NgClone ci = sortedNgCloneDB.get(i);

  // search connection targets
  for (j = i; j < ngCloneTotalCount; j++)
  {
    NgClone cj = sortedNgCloneDB.get(j);

    // Ng クローン ci と cj が近い距離にあれば
    dx = distance(ci.codeInX, cj.codeInX);
    dy = distance(ci.codeInY, cj.codeInY);
    if (((0 <= dx) && (dx < threshold2)) &&
        ((0 <= dy) && (dy < threshold2)))
    {
      // ギャップ情報生成
      Gap newGap = new Gap(ci, cj);
      gapDB.add(newGap);
    }
    else
    {
      // 最適化
      if (distance(c.codeInX, d.codeInY) >= threshold2)
        break; // 次の i ループへ
    }
  }
}

// 関数"distance(x, y)"では，コード片 x の終了位置と
// コード片 y の開始位置の距離を算出する．
// もし，x と y がオーバーラップしていたり，y が x より前方に
// 存在すれば，その結果はゼロもしくはマイナスの値をとる．
```

図 10: ギャップ検出アルゴリズム

ンから閾値 2 以内の距離にある Ng クローンは，高々定数個以内であると考えられる．したがって，内側の for ループは定数時間で終了し，アルゴリズム全体の計算複雑さは，$O(n)$(n: Ng クローン数) である．

また，閾値 2 の値の決定方法について議論する必要があると考えられるが，本研究においては定数値として扱うものとする．よって，閾値 2 は認識するギャップの最大長となっている．

本例においては，閾値 2 は 3 と定める．したがって実際は，g2，g4，g6 は検出されて

こない．

ステップ **3** (視覚化)

ステップ **3-1** (Gap-and-clone 散布図)

Ng クローンとギャップを散布図に描画する．

本ステップまでにおいて，Ng クローンとギャップの位置情報を得ることができたので Ng クローンとそれらのギャップをそれぞれ独立に散布図上に描画すれば，図 11(a) のように擬似的に Gapped クローンの視覚化を行うことができる (以降，本散布図を**Gap-and-clone** 散布図と呼ぶ)．

Gap-and-clone 散布図の両座標軸は，各コード列を表しており，左上角が原点になっている．図 11(a) では，垂直軸がコード片 X，水平軸がコード列 Y を表しており，各 Ng クローンは，両座標軸上で対応した 2 つの要素が一致していることを表している黒い四角形を連結した線分として表されている．また，各ギャップは，両端の Ng クローンを結ぶ灰色の線分として表現されている．つまり，黒い線分と灰色の線分を順に辿ったものが，Gapped クローンを表していることになる．表 3 に，そのパス (辿り方) とその対応コード列を示す．但し，g3，g4 は認識されなかったので，それらを通る辿り方は示さない．

ステップ **3-2** (フィルタリング)

Gapped クローンに関係のない **Ng** クローンとギャップを取り除く

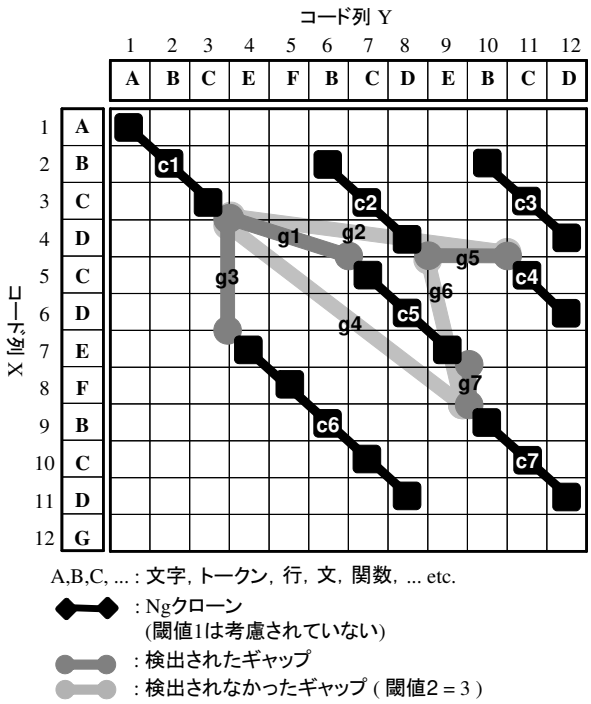
本ステップは随意的なものであるが，Gap-and-clone 散布図の視認性をかなり改善することができる．

図 11(a) には，コード片 X，Y から検出され得る全 Ng クローン (c1，...，c7)，全 gap(g1，...，g5) が描かれている．しかし，先に述べたように大規模ソフトウェアを対象とした場合，短い Ng クローンは非常に数多くの存在するので，Gap-and-clone 散布図上には多くの短い Ng クローンやギャップが存在し，分析が煩雑になる．

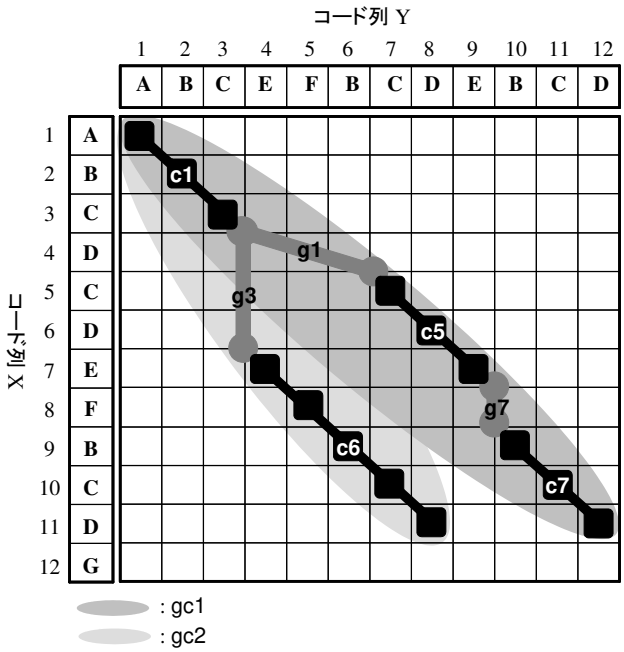
表 3: Gapped クローン

ID	パス	コード列 X	コード列 Y
gc1	c1 g1 c5 g7 c7	“ABC-CDE-BCD”	“ABC---CDEBCD”
gc2	c1 g3 c6	“ABC---EFBCD”	“ABCEFBCD”
gc3	c2 g5 c4	“BCDCD”	“BCD--CD”

(“-” はギャップを表している．つまり，それぞれの “-” の数がギャップの長さであり，そのギャップの両端の Ng クローンの距離となる．)



(a) フィルタリング前



(b) フィルタリング後

図 11: Gap-and-clone 散布図

閾値 1(Ng クローン最小一致長) と閾値 2(最大ギャップ長) を用いれば，Ng クローンとギャップの数を制御することができる．しかしこれらの閾値では，Gapped クローンとしての長さを制御することはできない．

例えば，閾値 1 を 3 にすれば，c4 , g2 and g5 を除いた全ての Ng クローンとギャップが現れる．逆にもう少し大きな値，5 にすれば，Ng クローン c6 しか残らない．

そこで，Gapped クローンを Gap-and-clone 散布図上でフィルタリングするため，各 Ng クローンの連結集合の長さを新しいパラメータとして導入する．その長さとは，連結集合中に含まれ得る Gapped クローンの上限長である．

連結集合中に含まれ得る Gapped クローンの上限長を算出手順は，まずギャップ検出の際に生成した，ギャップと Ng クローンの結合情報を用いて Ng クローンを連結集合に分類する．そして，各連結集合 s の中で，各コード片がコード列 X(Y) において最も先頭に近いコード片の開始位置を s の開始位置 sStartX(sStartY) とし，同様に最も先頭から遠いものを s の終了位置 sEndX(sEndY) として求め，次式

$$\texttt{sSize} = max(\ \texttt{sSizeX}\ ,\ \texttt{sSizeY}\ ),$$

$$\texttt{sSizeX} = \texttt{sEndX} - \texttt{sStartX},$$

$$\texttt{sSizeY} = \texttt{sEndY} - \texttt{sStartY}$$

で求まる sSize が連結集合中に含まれ得る Gapped クローンの上限長となる．そして，この長さの下限値を閾値 **3** とし，閾値 3 以上の長さを持つ連結集合のみを表示させることでフィルタリングを実現する．本例において，閾値 3 を 8 とするならば，Gap-and-clone 散布図は図 11(b) のようになる．

ステップ **4** (ソースコード分析)

各パラメータの値を調整しながら，**Gap-and-clone** 散布図を利用してソースコードの分析を行う.

最後に，Gapped クローンが擬似的に現れている Gap-and-clone 散布図を用いてユーザが実際に分析を行う．そのため，Gap-and-clone 散布図は，分析者に指定された Ng クローンやギャップの位置情報を提供するだけのインターフェースは備えているものとする．分析者は Gap-and-clone 散布図上で興味ある Gapped クローンが存在する範囲を絞りこみ，それらに含まれる Ng クローンの位置情報をインターフェースを通して知り，対応したソースコードを何らかの手段で実際に参照できればよい．

実際に Gap-and-clone 散布図上の Gapped クローンや，対応ソースコードを参照して，興味対象が見つからない場合は，パラメータを調整し直す必要がある．パラメータは，

以下の 3 つである．

閾値 **1** Ng クローン検出における Ng クローンの最小一致長，

閾値 **2** ギャップ検出におけるギャップ上限長，

閾値 **3** Ng クローン連結集合の下限長．

これらのパラメータは計算コストと完全性のトレードオフである．またその閾値毎にその効果は異なる．例えば，閾値 1 を小さくすると解析時間が増大する．しかし，数多く修正が施されたような，それぞれの一致部分が短い Gapped クローンを検出しようと思えば，閾値 1 に小さな値を定めなければならない．しかし，メソッドの抽出のようなリファクタリングなどが目的などの場合は，一般的にそのような Gapped クローンを 1 つにまとめるのは困難であると思われるので，さほど小さな値にする必要はないと考えられる．また，閾値 2 は大きくすると，1 つの Ng クローンから連結される Ng クローンの数が増えるため，解析時間が増大する．ソースコード長に近づくにつれ，計算複雑さは $O(n^2)(n$: Ng クローン数$)$ に近づく．

本アプローチの優位性は，個々の Gapped クローンを検出せずとも，ユーザは Gapped クローンがどこにあるのか識別することができるという点にある．仮に，個々の gapped-clone を全て検出した場合，Ng クローン連結集合 (グラフ) それぞれにおいて全ての辿り方を調べることになるため，計算複雑さは，最悪 $O(m!\ n)(m$: 各連結集合内に含まれる Ng クローン数，n: 連結集合数$)$ になる．それに対し，本アプローチにおいて，最もコストが高いのは，Ng クローンのソートだけであるため，$O(n\ \log\ n)(n$: 全 Ng クローン数$)$ の計算コストで押さえられている．

3.3 コードクローン情報の利用とそれに求められる抽出

3.3.1 利用目的

近年，コードクローンに関する研究は，我々同様，コードクローン検出手法の提案自体から，検出されたコードクローン情報の活用法やコードクローンの持つ性質の評価へとフォーカスを移しつつある．Lagüe らは，ソフトウェア保守の観点からメソッド単位でのコードクローン検出の有効性について評価し [35]，Baxter らは，発見されたコードクローンをマクロやルーチンとして定義して個々のコードクローンをマクロやルーチンの呼び出しに置き換えることでシステムの再構築を行っている [12]．Antoniol らは，同一システムの複数バージョンを比較することで，ソフトウェア進化の様子を知り，予測する手法を提案している [5]. また，山本らはコードクローン情報を利用してソフトウェアの類似度比較を行い [48]，門田らは，コードクローンとバグの関係を統計的に調べている [38]．

我々は，3.1 節における CCFinder の適用の中で，上記利用法を含め，開発保守プロセスにおいては，開発者，保守者，管理者の 3 者の観点から，次のようなコードクローンの分析・利用目的があると考える．ここでいう管理者とは，ソフトウェアの開発保守プロセス全体を管理，監視を行う者を指す．

1. 開発者的観点

利用目的 **1** プログラム記述ヘルプとしての利用

ある処理を記述している際，もしくはバグへの対処を行っている際に，その箇所をサンプルコードや既存ソースコードと比較することにより，類似コード片から処理方法のヒントやバグへの対応策を得られる可能性がある．

利用目的 **2** デバッグへの利用

実際にバグの位置を特定するのに利用するは困難であると思われるが，1 箇所バグが見つかり，修正を行ったが同様の修正を行うべき箇所の発見には利用できる．バグ位置の特定としては，コピーとペーストにより生成されたコード片における識別子名等の修正し忘れなど単純ではあるが，同一システム内でのクローンペアを調査すれば発見できる可能性もある．単純とはいえ，よく起こりがちなケアレスミスであり，例えば，ペースト先の名前スコープ内に同一名が存在していたならばコンパイル時にエラーとして検出されず，見つけにくいバグとなっている可能性がある．全てを調べ尽くすことは現実的に不可能であり効率もよくないが，ある程度までバグ位置が絞り込めていれば，バグ位置を特定できるかもしれない．

2. 保守者的観点

利用目的 **3** リファクタリングへの利用

2.1.2 節で述べたように，重複したコードに対するリファクタリング手法は様々ある．2.1.2 節では，オブジェクト指向言語を前提に述べたが，当然，メソッドの抽出などは，コードクローンを新たな関数，マクロ等のサブルーチンとして定義し，メソッド同様，個々のコードクローンをサブルーチン呼び出しに置き換えればよい．クラスの抽出については，汎用的によく使われる定型処理を部品化，ライブラリ化するとすると考えることができる．また，重複した条件記述のコード片の統合は，オブジェクト指向言語に限ったことでないし，繰り返し書かれた処理をループ化するということもできる可能性がある，

利用目的 **4** プログラムテキスト修正時のチェック

デバッグや，機能追加などにおける修正を行う際，同様の修正を行うべき箇所の発見をすることができる．

3. 管理者的観点

利用目的 **5** 設計品質評価への利用

対象ソフトウェアの中にどの程度類似しているものが存在するのかを全体的に確認することで，類似モジュールが数多く存在するかどうか，設計品質が劣化していないかどうかを把握することで，設計の見直しの必要性を知ることができる．例えば，多様性を保持すべき多重化された冗長系システムにおける多様性の評価などがある [31]．

3.3.2 利用に求められる分析・抽出

前節での利用目的のそれぞれに対し必要な抽出は次の 3 つに分類できる．

抽出 **1** 特定コード片に対するコードクローン

利用目的 1，2，4 に対しては，基本的に抽出のキーとなるソースコード片が与えられる．つまり，コードクローンの検出結果を開発者あるいは保守者が特定した位置に関するコード片に絞りこむ手段，環境さえ提供されればこれが単なる文字列検索と異なるのは，CCFinder 等においてプログラム言語の特徴が考慮されている点において異なる．但し，本研究においては，利用目的 2 を利用目的 4 と同等，バグの位置が特定されているものとみなす．

抽出 **2** リファクタリングに適したコードクローン

リファクタリングに適したコードクローンを抽出するため，利用目的 4 には，リファ

クタリングをしやすいかどうか，リファクタリングによってどのくらい効果があるかを考えなければならない．

まず，リファクタリングをしやすいかどうかは，2.1.2 節でのリファクタリング手法の多くがメソッド単位の手法であったように，メソッドレベルやある一連の機能的なまとまりをもったコード片としての類似が見られれば，直接的にリファクタリングを行いやすいと考えられる．逆に，一連の機能的なまとまりをもっていないと，リファクタリングを行ったとしてもプログラムの構造が複雑になる可能性もある．

また，コードクローンが存在するクラスやモジュールが，システム設計上どのような関係をもっているかによってもその行いやすさは変わってくる．例えば同一クラス内 (ファイル内) に閉じていたりすれば，比較的容易にリファクタリングが行えるであろう．逆に，様々なクラスに分散していた場合，スーパークラスへ引き上げるのがよいのか，新たなクラス，部品として委譲させるのがよいのか判断する必要がある．

そして，2.1.2 節で述べたように，リファクタリングの目的は，理解や修正が簡単になるようにすることである．したがって，リファクタリングの効果は如何に変更をしやすくするかで評価される．変更容易性が向上するかどうかにはついては，複雑さを如何に下げることができるかが問題になると考えられる．

そこで我々は，

- 機能的なまとまり，
- 存在位置間の関係，
- 複雑さ

の特徴を捕らえたコードクローンの抽出を考える．

抽出 **3** ファイル (モジュール) の類似性

利用目的 5 においては，対象ソフトウェア全体の中でどのあたりのモジュールにコードクローンが集中して存在しているのか，もしくはどの程度類似しているかが分かればよい．

まず，抽出 1 に関しては，基本的に CCFinder の解析結果からキーとなるコード片に対するコードクローンの検索機能，もしくは，はじめからそのコード片に対するコードクローンしか検出しない機能が提供されることで解決可能である．抽出 2 に関しては，3.3.3 節にて機能的なまとまりの抽出，3.3.4 節にて，存在位置間の関係，複雑さを考慮したメトリクスによるコードクローン抽出の 2 つを考える．抽出 3 については，3.1 節で述べたように既存の参照支援システムにおいて識別可能である．詳細については，4 節にて述べる．

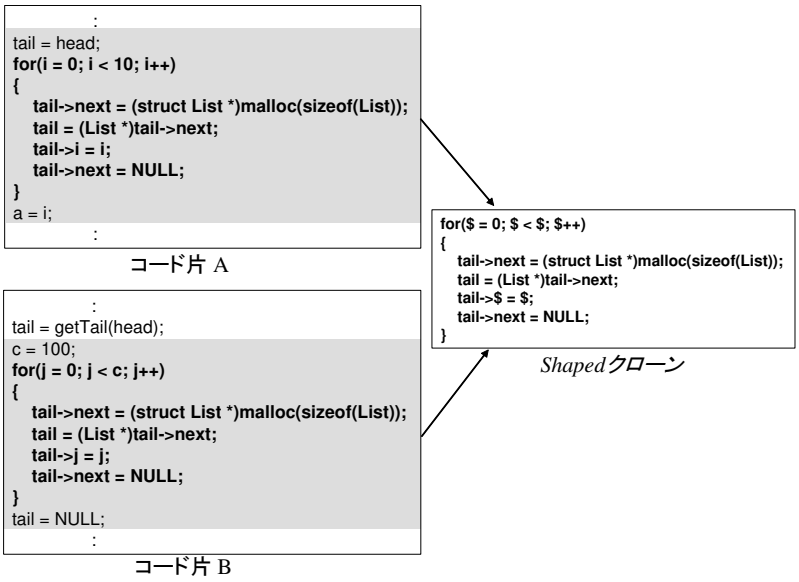


図 12: Shaped クローン

3.3.3 機能的なまとまりを持ったコードクローンの抽出

本来，プログラムテキストの意味や機能に言及するためには，ソースコードの構文解析，意味解析情報の直接的な利用が必要であると考えられる．しかし，一般的に構文解析，意味解析の計算コストは高い．例えば AST を利用したコードクローン検出ツール CloneDR[12] では 100KLOC のソースコードの解析に 2 時間かかり [27]，PDG を用いたコードクローン検出手法 [33][34] においては，大規模ソフトウェアへの適用可能性が課題となっている．CCFinder においては，構文解析や意味解析を行わないため，高速であり大規模ソフトウェアへの適用可能性を持っている．したがって，この大規模ソフトウェアへの適用可能性を失うことなく，構文解析や意味解析を行わずに機能的なまとまりをもったコードクローンを抽出することを前提とする．

プログラムテキストにおける意味は，C/C++や Java においては，クラス，メソッド (関数)，コンパウンド・ブロック，文等，Fortran においても，関数，文，COBOL においては，部 (division)，節 (section)，段落 (paragraph)，文等という様々なプログラム構造毎に，ある程度の一連のまとまりをもっていると考えられる．したがって，意味的・機能的な一致を見ないまでも，その構造さえ抽出することが可能であれば，コードクローンを機能的なまとまりに抽出できると言える．

例えば，図 12(C/C++) のようなコード片 A，コード片 B の比較を考えた場合，図中の網掛け部分が CCFinder からコードクローンとして検出される．

コード片 A は，リスト構造へ先頭から何かしらのデータを順に格納する処理，コード片 B は，リスト構造へ末尾から何かしらのデータを順に格納する処理となっている．これらの 2 つの間には，リストデータ構造を操作するという共通のロジックが `for` ブロックの中に含まれている．逆に，コードクローンとして検出された `for` ブロックの前後の文は，偶然の一致である．したがって，リファクタリングの観点からは，この前後の文はコードクローンから省かれた方がよい．

また，クラスやメソッドでは少し粒度が荒く，逆に文では，効果的な結果は期待しがたいため，我々は本研究において，機能的なまとまりの粒度としてコンパウンド・ブロック，そして抽出対象言語を C/C++，Java として考える．

コンパウンド・ブロックの抽出であれば，特別な手法は必要なく，字句解析を行いスタック等を用いるだけで容易に”{”と”}”の対応関係抽出可能である．

3.3.4 メトリクスを用いたコードクローンの抽出

本節においては，コードクローンの存在位置間の関係と，コードクローンの複雑さを考慮したコードクローンの抽出を考える．本節においても，3.3.3 節のような理由から，構文解析や意味解析を行わずに比較的低いコストでの抽出を前提とする．

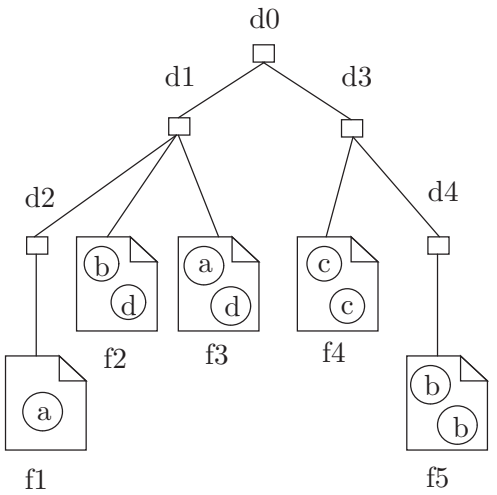
まず，コードクローンの存在位置間の関係に関しては，設計上のモジュール階層構造における距離や，継承関係，利用関係などが考えられるが，それらの情報を取得するには，ある程度以上の構文解析が必要になる．

一方，コードクローンの存在位置間の関係に関して，これまで文献 [30] において次のメトリクスが提案されている．

$RAD(C)$ (Radius)

クローンクラス C 内のコード片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．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.

図 13 を用いて簡単な例を示す．図中で各葉はファイルであり，その他のノードはディレクトリを表す．ファイル中に含まれる記号 a，b，c，d は各コード片が属するクローンクラスの識別子を表す．クローンクラス a について共通の親ノードは d1 であり，d1 までの距離が最も長いのは f1 であるから $RAD(a)$ は 2 である．クローンクラス b についても同様に 3 となる．しかし, クローンクラス c のように，単一ファイル内で閉じているクラスについては 0 と定義される．また，クローンクラス d のように同一ディレクトリ内のファイル集合内で閉じている場合は 1 となる.



□ ディレクトリ

ファイル

○ コード片

クローンクラス C	RAD(C)
a	2
b	3
c	0
d	1

図 13: クローンクラスの分散例

本メトリクスは，コードクローンが存在するファイルパスのみから容易に計測可能であるうえに，ディレクトリ構造はある程度のモジュール階層構造を表しており，リファクタリングのしやすさを測るには有効であると考えられる．

次に，複雑さについてであるが，一般にソフトウェアの複雑さとは，理論的複雑性と心理的複雑性に大別される [47]．心理的複雑性はソフトウェアプロセスの性質を反映したものであり，本研究において考慮しない．さらに理論的複雑性に計算の複雑性とプログラムの複雑性があるが，設計やプログラミングの結果が複雑性として現れたものは，プログラムの複雑性である．このプログラムの複雑性には，

ボリュームメトリクス

コード行数，ソフトウェアサイエンス理論 [23]，ファンクションポイント [2] など，

データ構造メトリクス

データの量，データ種類別の出現数 [14]，データの活性区間 [18]，データの参照間隔 [19] など，

制御構造メトリクス

サイクロマチック数 [37]，最大交差数 [15]，ノット数 [46]，ネストの深さ [50] など，

もしくは，それらの複合メトリクスがある．また，文献 [30] では，コードクローンに関するメトリクスとして，$RAD(C)$ に加えさらに次の 3 つのメトリクスが定義されている．

$LEN(C)$ (Length)

クローンクラス C 内の 1 コード片の長さを表す．

$POP(C)$ (Population)

クローンクラス C 内の要素数 (コードフラグメントの数) を表す.

$DFL(C)$ (Deflation)

クローンクラス C を再構築した場合に, どれだけコードサイズが減少するかという予測値を表す．ここでの再構築とは，クローンクラス C 内のコード片集合を新たな 1 サブルーチンにマージし，各箇所でコールするという最も単純な場合を考えている．一回のサブルーチンコールに必要なコードサイズを $callSize$ として DFL(C) は以下の式で定義される．

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

これらのメトリクス値は全て低いコストで計測することができる．また，上記分類では $LEN(C)$ はボリュームメトリクス，$DFL(C)$ は複合メトリクスに分類される．

3.1 節で，我々は既にコードクローンに対する単純な評価尺度を用いた CCFinder の解析結果の参照支援システムを試作したと述べたが，事実，これらのメトリクス ($RAD(C)$，$LEN(C)$，$POP(C)$，$DFL(C)$) の計測を行い，コードクローンの識別を試みた．しかし，ひとつの要素のコードサイズが大きいクローンクラス，要素数の多いクローンクラス，再構築後のコード減少予測値が大きいクローンクラスが，必ずしもリファクタリングの効果が期待できる複雑さを持ったものではなかった．例えば，変数宣言が多く繰り返されていたりすると，コードサイズとしては大きなコードクローンとなり，構文が単純であるが故に多く検出される．つまりこれらの単純なメトリクスのみではリファクタリングを行ったとしても効果が期待できないであろうコードクローンばかりが際立ってしまう場合があった．

そこで，我々はクローンクラス C に対し，さらに次の 5 つのメトリクスを定義する．

$LNR(C)$ (Length of non-repeated code)

クローンクラス C 内の各要素に含まれる非繰り返しコードサイズの最大値．“繰り返しである” コードの定義を例を用いて説明する．

例えば，コード列中で

```
X A B C A B C A B C A B C Y
```

のように 3 つの記号 A，B，C が繰り返していたとき，2 つ目の繰り返しの最後の記号から “繰り返し” であると判定する，すなわち，以下の “*” が付された部分を繰り返しであると定義する．

```
X A B C A B *C *A *B *C *A *B *C Y
```

これは一見，

```
X A B C *A *B *C *A *B *C *A *B *C Y
```

のように定義する方が合理的であるように考えられる．しかし，この場合，もしコードクローンとして検出されたコード片が C A B であった場合，

```
X A B [C *A *B] [*C *A *B] [*C *A *B] *C Y
```

のように，このコード列中から 3 のコード片 (“[” ，“]” で囲まれた部分) が，C A B を各要素とするクローンクラスに含まれる．すると，これら 3 つの各コード片の非繰り返しコードの長さは，左から順に 1，0，0 と判定される．

一方，

```
X A B C A B *C *A *B *C *A *B *C Y
```

であれば，もしコードクローンとして検出されたコード片が C A B であっても，

```
X A B [C A B] [*C *A *B] [*C *A *B] *C Y
```

となり，非繰り返しコードの長さは，左から順に 3，0，0 と判定される．$LNR(C)$ は “繰り返し部分に含まれるコード片からコードクローンを検出しないようにする” 目的で使われるため，この例では，少なくとも先頭の C A B については，このように非繰り返しコードの長さを 3 と判定することが望まれる．もちろんコードクローン片の意味的な先頭になり得る記号が区別可能であれば，こういった問題は起こらない．しかし，現実的に人間が見てもすぐにはコードクローン片の最初を判断しかねる場合もある．例えば，

```
   :
代入文
出力文
代入文
出力文
   :
```

のようなコード列があった場合，2 つ目の代入文は出力文の後始末をしているのか，次の出力に備えた処理なのか判断は難しい．ゆえに，このような定義となっている．$LEN(C)$ と異なり，単なるコード長ではないボリュームメトリクスである．

現在，CCFinder はコードクローン検出時に同時に各コード片の非繰り返しコード長を算出可能である．

$RNR(C)$ (Ratio of non-repeated code)

クローンクラス C 内の各要素のコード片の中で，非繰り返しコードのサイズが占める割合 (パーセンテージ)．つまり，

$$RNR(C) = \frac{LNR(C)}{LEN(C)} \times 100 \quad (0 \leq RNR(C) \leq 100)$$

で定義される複合メトリクスである．

$CMP(C)$ (Complexity)

クローンクラス C 内の各要素に含まれる条件分岐予約語数 (スイッチ文はひとつと数える) の最大値．いわゆるサイクロマチック数 [47] の近似を行うことで，制御構造の複雑さを測る．

特に，C/C++，Java に関しては，`if`，`for`，`while`，`switch` の予約語の数として定義する．2.1.2 節で述べたように，スイッチ文などの分岐は重複したコードを生成しやすい．しかし経験的に，重複コードを生成しやすいが故にスイッチ文周辺には長さが異なるクローンペアが大量に検出される可能性が高く，`case` をひとつひとつカウントするとスイッチ文周辺のみしか抽出できないことが予想される．そのため，条件分岐予約語数に `case` はカウントしない．

$USR(C)$ (Number of names of identifier defined by users)

クローンクラス C 内の各要素に含まれるユーザ定義名の種類の数の最大値を取る．ユーザ定義名の種類数が多ければ多いほど，個々の変数を識別したり，それらの間の関連を把握したりすることが難しくなる．データ構造メトリクスに分類される．

$RFT(C)$ (Refactoring)

クローンクラス C のリファクタリング効果の予測する．クローンクラス C に対する $RFT(C)$ を以下のように定義する．

$$\begin{cases} RNR(C) < 50 \rightarrow RFT(C) = 0 \\ RNR(C) \geq 50 \rightarrow RFT(C) = (\frac{LNR(C)}{maxLNR} \times 100 + 1) \times (\frac{CMP(C)}{maxCMP} \times 100 + 1) \\ \qquad\qquad \times (\frac{USR(C)}{maxUSR} \times 100 + 1) \times (\frac{POP(C)}{maxPOP} \times 100) \\ \qquad\qquad (0 \leq RFT(C) \leq 100,000,000) \end{cases}$$

ここで，$maxLNR$，$maxCMP$，$maxUSR$，$maxPOP$ は抽出対象全クローンクラスにおける各メトリクス値の最大値を意味する．本定義における基本方針は，

- 非繰り返しコードのサイズが占める割合が 50%未満 $(RNR(C) < 50)$ であれば繰り返しコードである可能性が高いため除外する．$RNR(C)$ が 50%未満のものは，繰り返しコードでループ化やメソッド抽出のリファクタリングをできる可能性も持っていることも考えられるが，経験的に，変数宣言の繰り返しコード等が検出された全コードクローンのかなりの割合を占めることは多く，実用的に意味をもたないものを除外することは重要である．
- ボリューム $(LNR(C))$，制御構造 $(CMP(C))$，およびデータ構造 $(USR(C))$ の複雑さが軽減されればリファクタリングの効果は大きいと考えられる．
- より要素数 $(POP(C))$ の多いクローンクラスをリファクタリングすることできれば，保守作業において 1 つの変更を多くの箇所に反映する必要性がなくなり変更容易性が向上する考えられる．
- メトリクス毎に値の単位と範囲が異なるので 0 以上 100 以下の値に正規化する (例：$\frac{LNR(C)}{maxLNR} \times 100$)．
- 各係数が 0 になった時，$RFT(C)$ として 0 にならないように各係数に 1 を加える．

となっている．

また，コードクローンの存在位置間の関係 $(RAD(C))$ は，リファクタリングのしやすさだけでなく，リファクタリングの効果にも影響してくると考えられる．例えば，直観的には，距離が離れれば離れるほど保守作業における変更が同時に反映されずバグとなる可能性が高くなりやすい．また，多くの要素が離れたところに (広く) 分散していればいるほど，様々なプログラムコンテキストに適合し利用価値のある，汎用性の高い部品が抽出できる可能性もある．しかし，逆に近ければリファクタリングがしやすいというリファクタリングのしやすさとリファクタリング効果のトレードオフの意義を含んでいるため，$RFT(C)$ に $RAD(C)$ は考慮しない．

以降，$RAD(C)$ は単に RAD と呼ぶ (他も同様)．

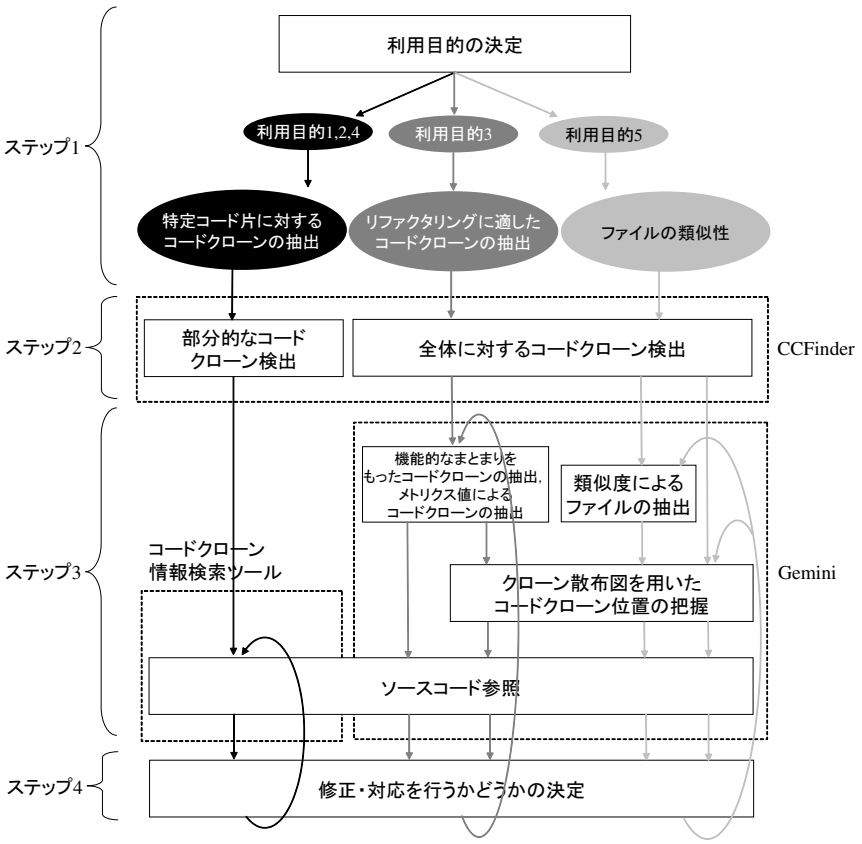


図 14: コードクローン情報利用プロセス

4　コードクローン分析支援環境 Gemini

本節では，開発保守支援を目指したコードクローン分析支援環境 Gemini の構築を行う．コードクローン情報の利用プロセスを考えた上で，システムの構成を設計し，実装を行う．

4.1　コードクローン情報利用プロセス

我々が考えるコードクローン情報利用プロセスは次の 4 つのステップで構成される．利用プロセス全体の流れを図 14 に示す．

ステップ 1 利用目的の決定

3.3.1 節で述べた 5 つの中から目的を選ぶ．それにより抽出すべきコードクローンが決定する．

ステップ 2 コードクローン検出

CCFinder を用い，コードクローンを検出する．特定のコード片に対するコードクローンの抽出を行いたい場合以外は，分析対象ファイル全ての間でのコードクローンを検出 (全体に対してコードクローン検出) を行えばよい．しかし，特定のコード片に対するコードクローンの抽出を行う際，分析対象ファイル同士の間でのコードクローン情報は当然必要ない．そのため，初めから分析対象ファイル同士での比較を行わないようにコードクローン検出 (部分的なコードクローン検出) を行う．実際，CCFinder において，あるファイル群とあるファイル群の間の比較のみを行い，各ファイル群内での比較は行わないという機能が備えられている．したがって，一方のファイル群を探したい特定コード片のみが書かれたファイルとして，他方を探索対象ファイル群として CCFinder を実行することにより容易に実現可能である．

ステップ **3** 目的のコードクローンを探す

まず，図 14 中で，特定のコード片に対するコードクローンの抽出を行う場合の “コードクローン情報検索ツール” については，文献 [28] で詳しく述べられている．したがってこれ以降，特定のコード片に対するコードクローンの抽出については述べない．それ以外の部分が本論文において支援の対象とされている部分であり，Gemini において実現する．

次に，リファクタリングに適したコードクローンの抽出を行う場合は，3.3.3 節，3.3.4 節において提案したとおり，機能的なまとまりをもったコードクローンの抽出，もしくはメトリクスを用いたコードクローンの抽出を行う．その後，クローン散布図 (4.2 節参照) などを用いて抽出されたコードクローン位置情報を確認し，ソースコードの参照を行う．

そして，ファイルの類似性の抽出を行う場合は，コードクローンからファイルの類似度を調べる，もしくはクローン散布図等を用いることで類似ファイルを探しのソースコードの参照を行う．ファイルの類似度に関しては，4.2 節にて述べる．

ステップ **4** 修正・対応を行うかどうかの判断

ステップ 3 において見つかったコードクローン，もしくはファイルが本当に利用目的に適したものであるかどうか判断する．例えばリファクタリングを行うかどうかについてである．もし，この際，意図しているコードクローンでなかったならば，ステップ 3 へ戻り目的のコードクローンを探すということが必要になる．

さらに，ステップ 4 で修正・対応が行われるべきだと判断され，修正・対応を行った場合，ステップ 1 もしくはステップ 2 へ戻り，再度コードクローンを検出する必要があり，このサイクルの繰り返しにより，目的が達成されるものと考える．

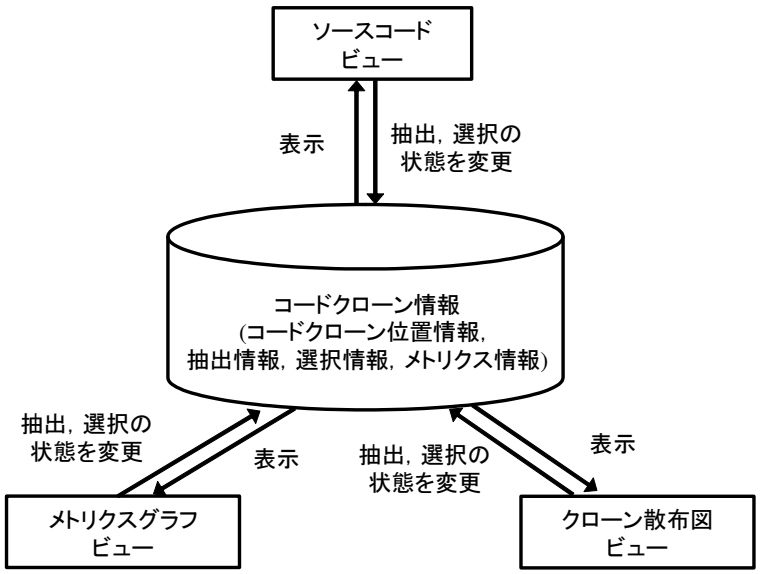


図 15: システムの構成

4.2 システムの構成

本節では，前節での利用プロセスを基に，システムの構成を設計する．

図 15 にシステム構成の概念図を示す．

図 15 では，中央に位置するコードクローン情報がデータモデルであり，それを 3 つのビューが取り囲んでいる．つまり MVC アーキテクチャにおけるコントローラは省略された形で書かれている．

利用プロセスを基に，本システムに必要なコンポーネントは単純に，機能的なまとまをもったコードクローン抽出コンポーネント，メトリクスによるコードクローン抽出コンポーネント，クローン散布図表示コンポーネント，ソースコード表示コンポーネント，ファイル類似度表示コンポーネントであると考えられる．

まず，機能的なまとまりをもったコードクローン抽出コンポーネントは，抽出された結果が CCFinder の解析結果と同様にクローンペアの位置情報であるため，図 15 の中において，コードクローン情報データモデルの中に含まれるものとされる．つまり，CCFinder の解析結果全てを更新してしまうものであるとする．メトリクス値も同様に CCFidner からのコードクローン情報からデータモデルが算出するものとする．

次に，メトリクスによるコードクローン抽出コンポーネントは図 15 の中において，“メトリクスグラフビュー”となっている．本ビューは，各メトリクス値をグラフで表示し，グラフ上での各メトリクス値に関して上限，下限の指定によりクローンクラスの抽出 (抽出状態の変更) が行えるだけの機能を備えたものとして定義する．

ここで，コードクローン情報に関してそれぞれ抽出状態と選択状態を定義する．抽出状態は，各クローンクラスについて定義され，各ビューにおいて表示されるのは，抽出状態にあるクローンクラス，もしくはそれに含まれるクローンペア，それに関連しているファイルのみに関する情報のみであるとする．選択状態はクローンクラス，クローンペア，ファイルに関して定義され，その選択状態はそれぞれ独立しているもとする．但し，選択状態になることが可能なのは，抽出状態のクローンクラス，抽出状態のクローンクラス内のクローンペア抽出状態のクローンクラス内要素を含むファイルであるとする．

そして，クローン散布図表示コンポーネントは，“クローン散布図ビュー”と対応するが，ここでいうクローン散布図は，両座標軸に共に同じソースファイル群のコード列が共に同じ順に並び，各座標位置において両座標軸における対応要素が一致していれば点をプロットするという図を指す．したがって，ある程度の大きさをもったクローンペアは，対角線分として図中に表れる．また，主対角線上の各点においては，常に要素の一致するため常に主対角線は描画され，各プロットの分布状態は主対角線に対して線対称である．さらに図中のクローンペアを選択状態にするだけの機能はもつものと定義する．

また，ソースコード表示コンポーネントとしては，“ソースコードビュー”であるが，それぞれの選択状態にあるクローンクラス，クローンペア，もしくはファイルのソースコード参照手段を提供する．クローンクラス，クローンペアの表示の際には，コードクローンがどのコード片であるか確認できる機能をもつものとする．

最後に，ファイル類似度表示コンポーネントは図 15 の中には現れてこないが，“ファイル類似度グラフビュー”として定義し，本ビューにおいては，文献 [44] において定義された $RSA(f)$ ，$RST(f1$,$f2)$ の値をグラフとして表示する機能をもつものであるとする．$RSA(f)$ は，ファイル f が f 以外の解析対象ファイルからコードクローンでどれだけカバーされているかという割合を指す．$RST(f1, f2)$ は，ファイル $f1$ がファイル $f1$ とファイル $f2$ との間のクローンペアでどれだけカバーされているかという割合を指す．

4.3 実装

本支援環境 Gemini は Java で実装されており (18KLOC)，JDK1.3 以上の VM が実行可能な環境で動作する．図 16，および図 17 に実際の Gemini のスナップショットを示す．

図 16 の中にはスナップショット上の GUI コンポーネントが何であるかを示す注釈が付されている．4.2 節で述べられていない，4 つのビュー，“ファイル一覧ビュー”，“クローンクラス一覧ビュー”，“選択状態クローンクラス内コード片一覧ビュー”，“クローンペア一覧ビュー”はそれぞれ，ファイル，クローンクラス，コード片，クローンペアに関して，“メトリクスグラフビュー”や “クローン散布図ビュー”によって得られない詳細な数値情報を示すものである．また，“メトリクスグラフビュー”は各メトリクスを 1 本の軸とした並行座標軸

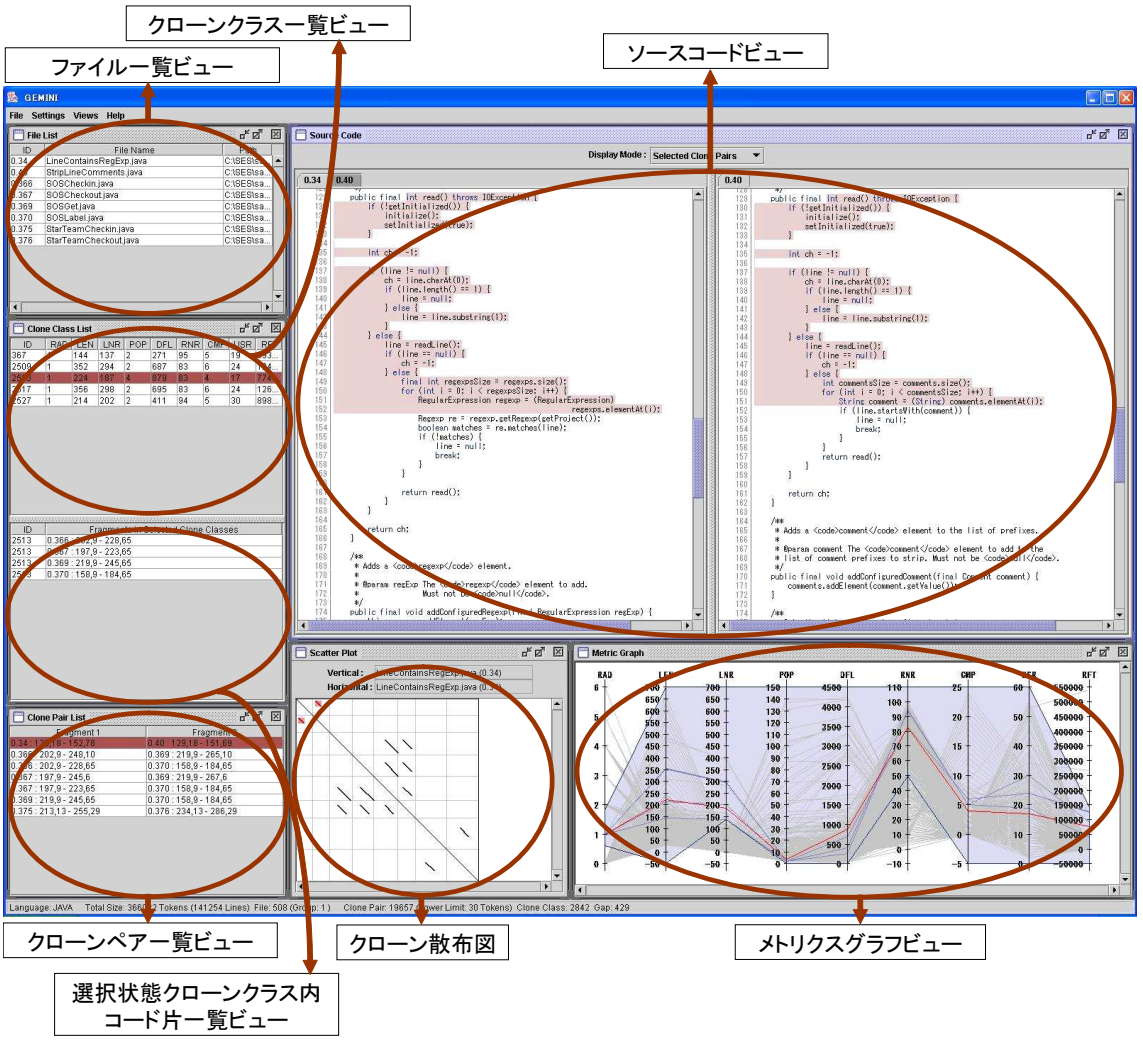


図 16: Gemini スナップショット

グラフとして実装されている．本グラフにおいては 1 クローンクラスにつき 1 本の折れ線が描かれる．

図 16 では，ファイル類似度グラフビューのみのスナップショットとなっている．上部には RSA 値に関するグラフ，その下には各ファイルに対する RST 値のグラフ順に表示されている．またソート機能を備え，あるファイルに対し最も類似したファイルを探す等の作業を効率的に行える．

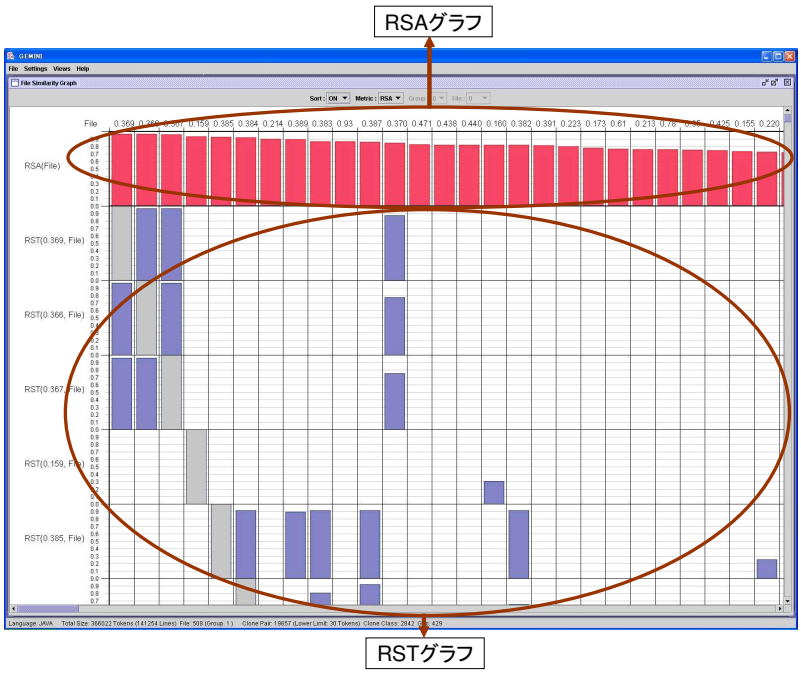


図 17: Gemini スナップショット (ファイル類似度グラフビューのみ)

5 評価

5.1 Gapped クローンの検出の適用実験

5.1.1 概要

本適用実験においては，大阪大学のプログラミング演習において学生が作成したプログラムのソースコードを対象とし，Gapped クローンの検出手法の適用を行う．

当該プログラミング演習において学生は，Pascal 風言語 (Pascal 言語のサブセット) で記述されたプログラムを CASL(アセンブリ言語) に変換するコンパイラを C 言語で作成する．また，その作成課題は，次の 3 つのステップ (副課題) で構成されている．

ステップ 1(課題 1): 構文解析器 ($Parser$) の作成.

ステップ 2(課題 2): 意味解析器 ($Checker$) の作成.

ステップ 3(課題 3): コンパイラ (SPC) の作成．

また，$Checker$ と SPC を作成するにあたり，各課題の前課題のプログラムを再利用することが課題として求められている．つまり，$Checker$ は $Parser$ を再利用することで，SPC は $Checker$ を再利用することで作成される．したがって，各課題のプログラム間の関係は，

我々が検出を想定しているコピーとペーストおよびその修正によって書かれたコードと同じと考えることができる．これが，適用実験の対象に選択した主旨である．

本適用実験に際し，69 人の学生のプログラム ($Parser$，$Checker$，SPC) を収集した (計 360KLOC)．

5.1.2 分析

本手法の有効性を確認するため，以下の項目について分析を行った．

(1)Gap-and-clone 散布図の有効性: Gap-and-clone 散布図の有効性を確認する意味で，閾値 1(Ng クローンの最小一致長) や，閾値 3(Ng クローン連結集合の下限長) を何度か変え，短い Ng クローンがある程度の大きさを持った Gapped クローンとしてどの程度現れてくるのかを確認する．

(2) どのような Gapped クローンが見つかるのか: Gap-and-clone 散布図で Gapped クローンとして見えているコードが実際にどのようなものであるのかを調べる．また図 8 のような Gapped クローンが本当に存在するのかどうかを確認する．

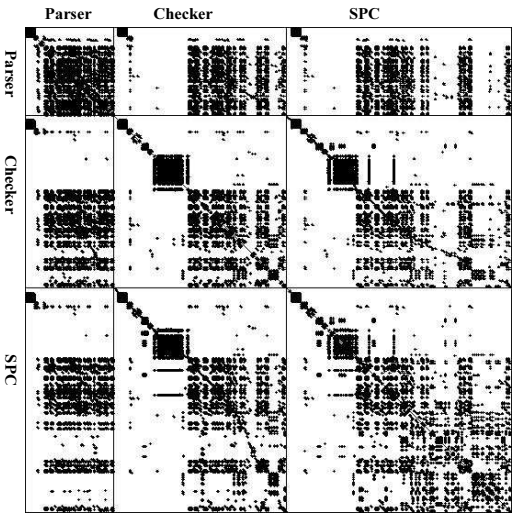
69 人のプログラムを対象に実験，分析を行ったが，[44] の課題間再利用率評価実験において，その評価が最も高かった学生 (S) を例にとって，その分析結果を示す．

(1)Gap-and-clone 散布図の有効性

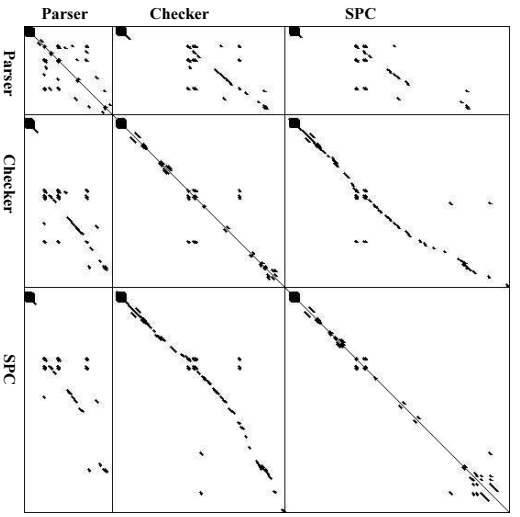
図 18 に S の $Parser$(2267 トークン)，$Checker$(4394 トークン)，SPC(5738 トークン) の 3 プログラム間の比較結果の散布図を示す．ここで Gap-and-clone 散布図にギャップを表示せず，かつ CCFinder からの検出された Ng クローンを全て表示する通常のクローン散布図を Ng クローン散布図と呼ぶことにする．

図 18(a) の Ng クローン散布図では，10 トークン以上の Ng クローンが全て描かれている (閾値 1=10 トークン)．図中に膨大に存在する黒い点は，非常に多くの短い Ng クローンが存在することを意味しており，つまりは，仮に全 Ng クローンを調べるとするならば，きめの細かい分析が可能である．また，少し大きな Ng クローンのみを調べるため，閾値 1 を 30 トークンにすると Ng クローン散布図は図 18(b) のようになる．この図には，30 トークン以上の長めの Ng クローンしか現れていないので，容易に分析が行える．しかし，コピーとペーストとその修正によって生じたコードクローンなどは多く抜け落ちている可能性がある．

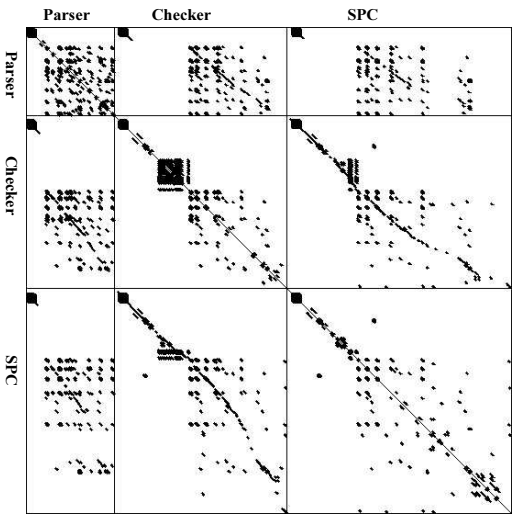
それに対し，Gap-and-clone 散布図は，個々の Gapped クローンを検出することに比べて計算コストもかからずに，きめの細かい分析が容易に行える．図 18(c) は，閾値 1 を 10



(a) Ng クローン散布図 (閾値 1 = 10 トークン)



(b) Ng クローン散布図 (閾値 1 = 30 トークン)

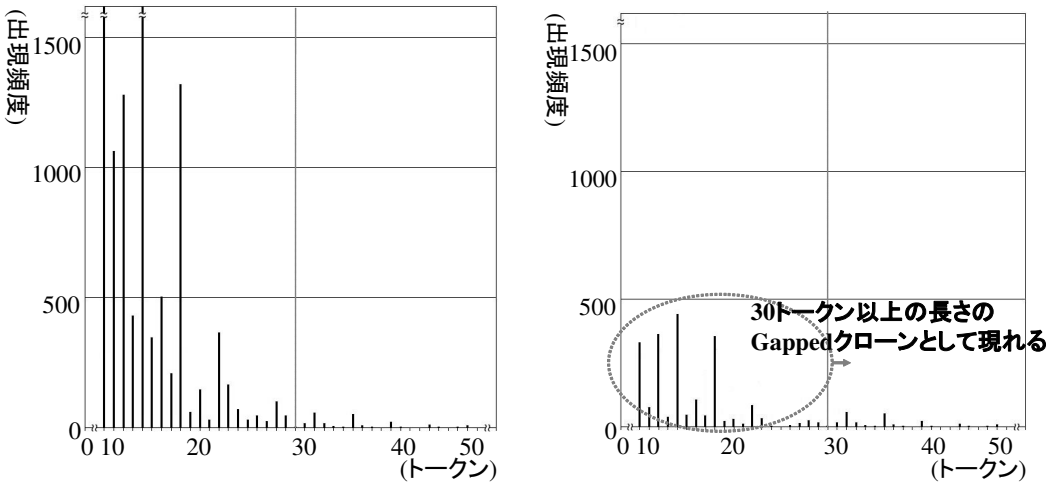


(c) Gap-and-clone 散布図 (閾値 3 = 30 トークン)

図 18: Gap-and-clone 散布図と Ng クローン散布図の比較

トークン，閾値 2 を 10 トークン，閾値 3 を 30 トークンとして S の 3 プログラムを比較した Gap-and-clone 散布図である．つまり，10 トークン以上の Ng クローンと 10 トークン未満のギャップで構成された 30 トークン以上の Gapped クローンが現れている．

Ng クローン散布図においての分析のきめの細かさと容易さとの間でのトレードオフと，Gap-and-clone 散布図によるそれらのトレードアップを定量的に説明する．図 19 の棒グラフには，Ng クローン散布図上の Ng クローンの出現頻度と Gap-and-clone 散布図上の Ng ク



(a) 図 18(a)(b) における Ng クローン出現頻度　　(b) 図 18(c) における Ng クローン出現頻度

図 19: 長さ毎の Ng クローン出現頻度

ローンの出現頻度が表されている．

図 19(a) の棒グラフは，図 18(a) の Ng クローン散布図に含まれる Ng クローンはほとんどが 10 トークンから 30 トークン未満であり，図 18(b) の Ng クローン散布図には，少ししか Ng クローンが現れていないことを示している．逆に，図 19(b) の棒グラフでは，約 2000 個近い 10 トークンから 30 トークンの Ng クローンが 30 トークン以上の Gapped クローンを構成するのに寄与していることが分かる．

これらのことは，特に学生 S に限ったことでなく，他の学生についても確認することができた．

(2) どのような Gapped クローンが見つかるのか

図 20 の 3 つのソースコードは，S の $Parser$，$Checker$，SPC にそれぞれ含まれる関数 “sentence” のソースコードである．また，図 20 の中には，それら 3 つのソースコードを対象とした Gap-and-clone 散布図 (閾値 1=10 トークン，閾値 2=10 トークン，閾値 1=20 トークン) が表されている．図中の記号 A，B，C，D，E は，散布図上のコードクローンとソースコード中の網掛け部分との対応関係を示す．

Gap-and-clone 散布図上の E には，3 つの Ng クローン E_1，E_2，E_3，およびそれらの間のギャップが存在する．これらのギャップは $Checker$ から SPC へ拡張する際に関数 “sentence” の機能拡張のために新規追加されたコード部分と対応している．これはまさに我々が検出を想定したコピーとペーストとその修正によって生成された Gapped クローンである．この

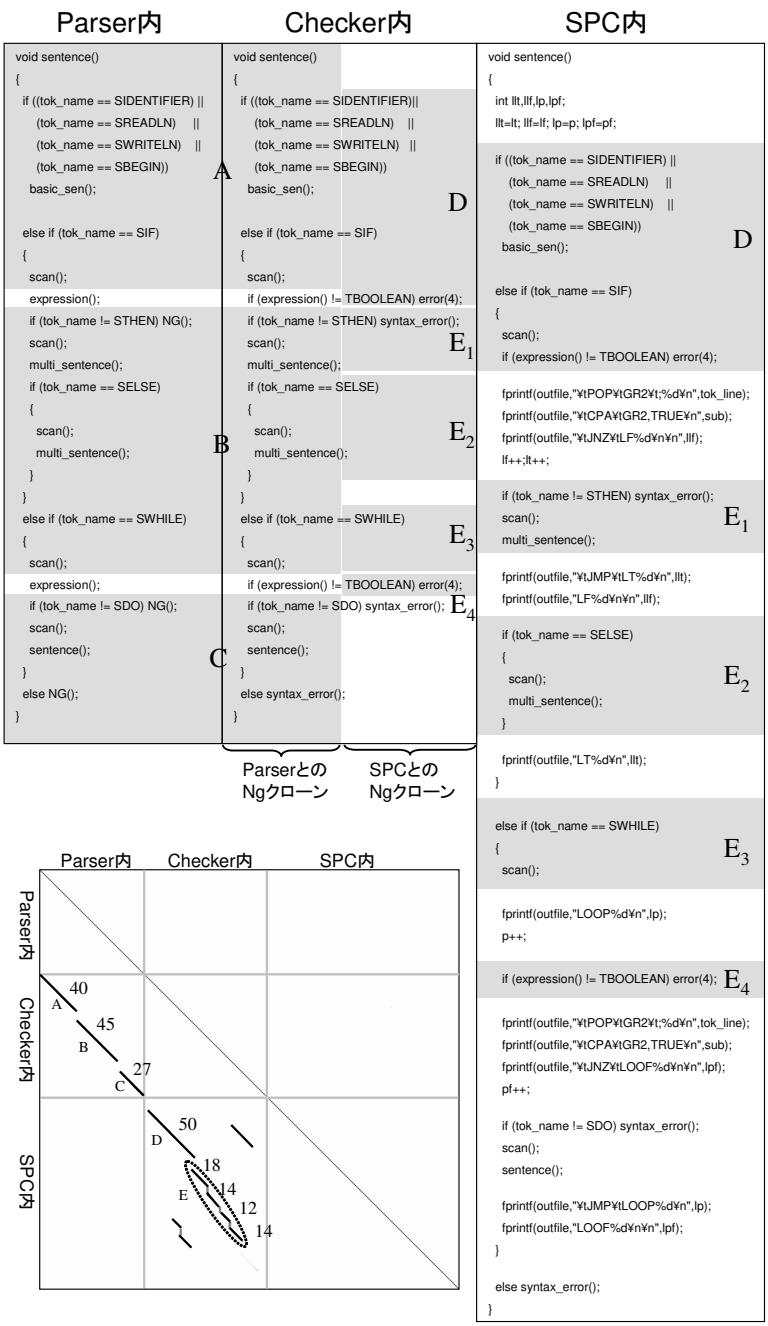


図 20: S のプログラム中の関数 “void sentence()”の各課題間での Gapped クローン

Gapped クローン E の中に含まれる個々の Ng クローンは 20 トークンに満たないため，単に 20 トークン以上の長さをもった Ng クローンを描画する Ng クローン散布図では現れてこない．

また，その他に見つかったGappedクローンとして，CCFinderにより分断されていた関数がGappedクローンとして連結されてしまったものが存在した．

その他の結果として，CCFinderでは，複数の関数をまたぐNgクローンが検出されないようになっているが(2.3.1参照)，ある関数の末尾付近とその次に並ぶ関数の先頭付近に存在する2つのNgクローンをGappedクローンとして再度結合してしまっているケースが多数見つかった．こういうケースは排除されなければならないが，これはGappedクローン連結の際に，各関数の区切り位置情報を利用すれば容易に解決できる．

5.2 機能的なまとまりを持ったコードクローンの抽出の適用実験

5.2.1 概要

機能的なまとまりを持ったコードクローンの抽出の有効性を確認するため，我々はJavaで書かれた次の2つのソフトウェアに本手法の適用を行った．

ANTLR[4] (ANother Tool for Language Recognition)

ANTLRとは，指定された文法記述からインタプリタやコンパイラを作るためのフレームワークを提供するツール(パーサジェネレータ)である．Java，C++，Sather の3つの言語を扱うことができる．

Ant[3]

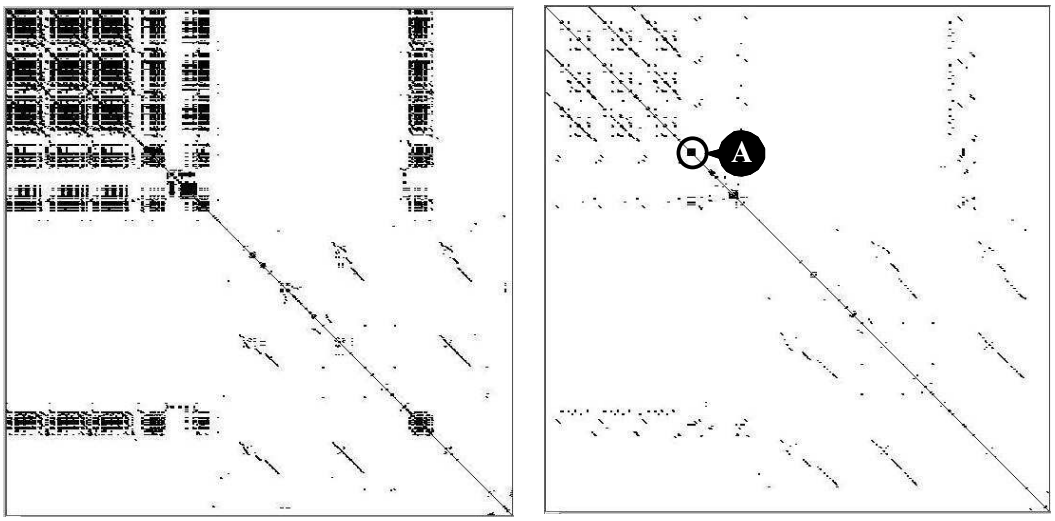
Antは，Javaベースのビルドツールであり，シェルコマンドを記述するかわりに，さまざまな実行タスクのターゲットツリーの呼び出しをXMLベースで記述することができる．

5.2.2 分析

本適用実験においては，CCFinderにおける最小一致トークン数を50トークンに定め，コードクローン検出を行った．

表4: ソースコードサイズ

	ファイル数	SLOC	トークン数
ANTLR	239	43548	140802
Ant	508	141254	221203



(a) 機能的なまとまりを持ったコードクローンの抽出を行わなかった場合

(b) 機能的なまとまりを持ったコードクローンの抽出を行った場合

図 21: クローン散布図 (ANTLR，最小一致トークン数 50)

(1)ANTLR への適用

ANTLR のソースコードは 239 ファイルで構成され，合計で 44KLOC となっている (表 4 参照)．図 21(a) のクローン散布図は，機能的なまとまりを持ったコードクローンの抽出を行わなかった場合のコードクローンの検出結果を示しているが，見ての通り ANTLR の中には非常に多くのコードクローンが存在する．クローンペア数としては約 34 万組，クローンクラス数としては約 1000 クラス存在する (表 5 参照)．これらの膨大な情報の中からリファクタリングを行うべきコードクローンを探し出すのは，非常に困難であると考えられる．

一方，図 21(b) のクローン散布図は，機能的なまとまりを持ったコードクローンの抽出を行った場合のコードクローンの検出結果を示している．散布図を見ても多くのコードクローンが切り捨てられ，クローンペア数は約 1000 組に，クローンクラス数は約 140 クラスに絞り込まれている (表 5 参照)．抽出を行わなかった場合と比較して，クローンペア数は $\frac{1}{350}$

表 5: 検出されたコードクローン数 (ANTLR，最小一致トークン数 50)

	抽出を行わなかった場合	抽出を行った場合
クローンペア数	338574	972
クローンクラス数	1072	142

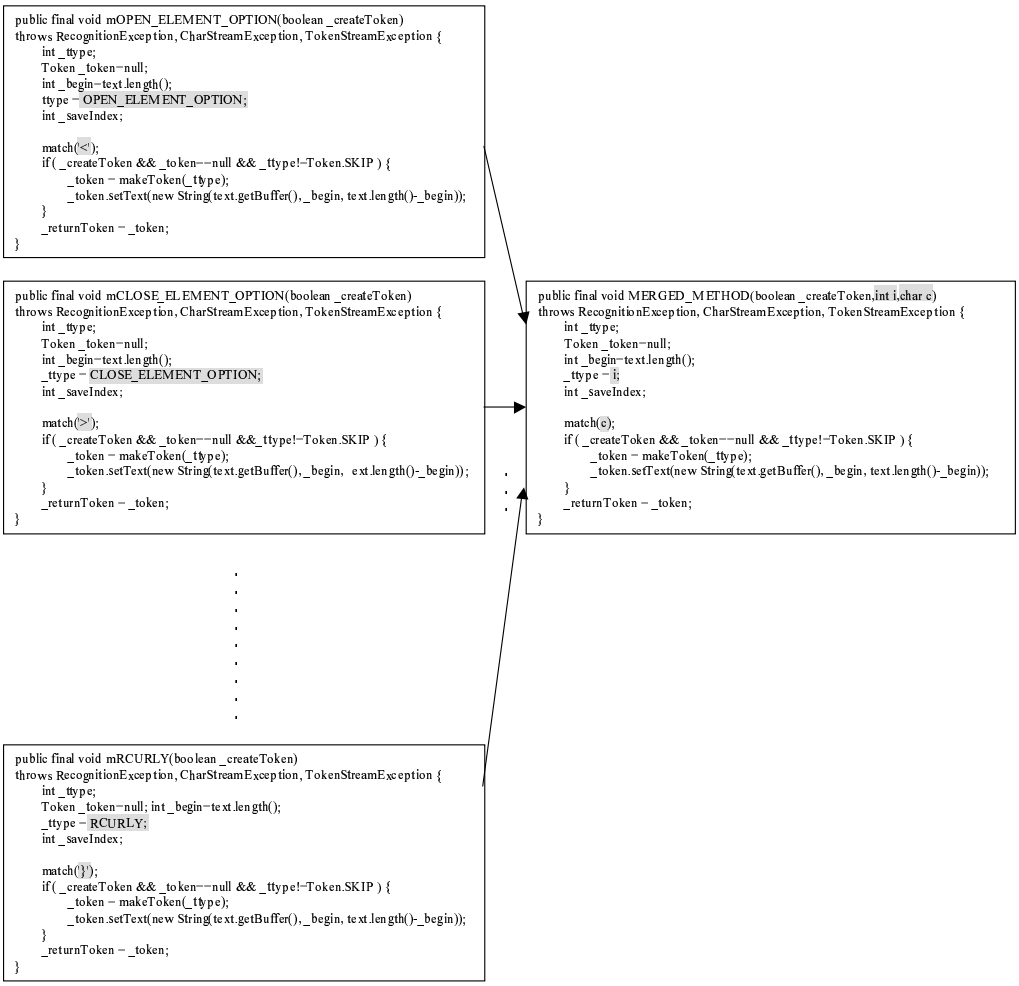
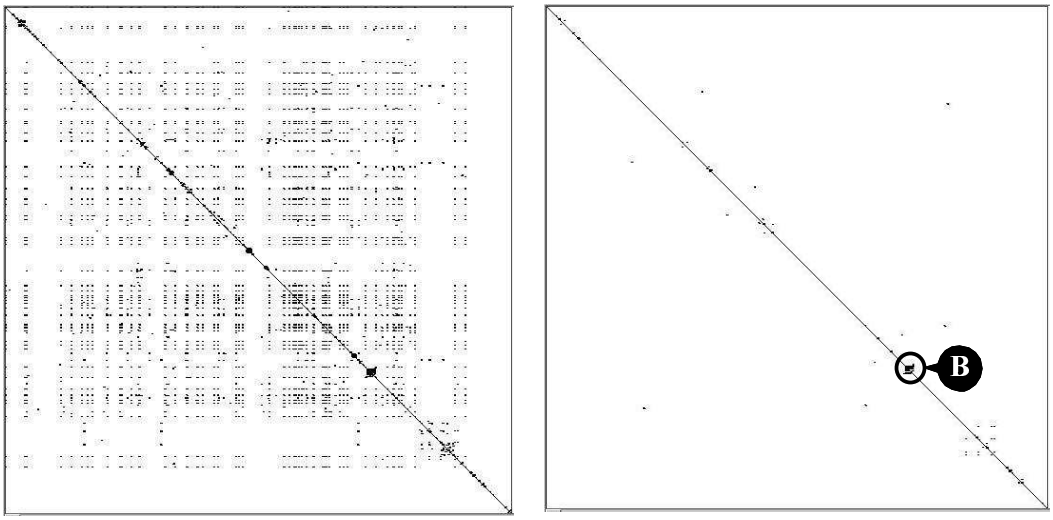


図 22: メソッド抽出の例 (ANTLR)

に，クローンクラス数は $\frac{1}{8}$ になっている．

そこで，我々は実際に，図 21(b) 中でコードクローンが一番密集している記号 A が付された箇所に対応したソースコードを参照した．A で囲まれた中には，28 組のクローンペアが存在し，そのそれぞれのコード片のサイズは 82 トークンであったが，これらのコードクローンは図 22 のように容易に一つのメソッドにまとめる (メソッド抽出する) ことが可能であることが確認された．図 22 中の左側のコード片は実際に存在したソースコードであり，右側は，それらをひとつのメソッドにまとめた場合の例である．

(2)Ant への適用



(a) 機能的なまとまりを持ったコードクローンの抽出を行わなかった場合

(b) 機能的なまとまりを持ったコードクローンの抽出を行った場合

図 23: クローン散布図 (Ant，最小一致トークン数 50)

次に，Ant への適用実験の適用を行った．Ant のソースコードは 508 ファイルで構成され，合計で 141KLOC となっている (表 4 参照)．図 23(a) のクローン散布図は，機能的なまとまりを持ったコードクローンの抽出を行わなかった場合のコードクローンの検出結果を示しているが，散布図全体に広く分散していることが見てとれる．クローンペア数としては約 12000 組，クローンクラス数としては約 850 クラス存在する (表 5 参照)．

一方，図 23(b) のクローン散布図は，機能的なまとまりを持ったコードクローンの抽出を行った場合のコードクローンの検出結果を示しており，クローンペア数は約 100 組に，クローンクラス数は約 50 クラスに絞り込まれている (表 5 参照)．見ての通り，ほとんどのコードクローンが切り捨てられて，記号 B が付された箇所のコードクローンが浮き彫りになっている．これらのコードクローンの実際のソースコードを参照してみたところ，これらは 7 つのクラス (ここでいうクラスはクローンクラスではない) に一つずつ存在した，完全に同一のメソッドであった．しかも各メソッドが存在する 7 クラスは全て同一のクラスを直接継

表 6: 検出されたコードクローン数 (Ant，最小一致トークン数 50)

	抽出を行わなかった場合	抽出を行った場合
クローンペア数	12033	103
クローンクラス数	856	53

```
public void getAutoresponse(Commandline cmd) {
  if (m_AutoResponse == null) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_DEF);
  } else if (m_AutoResponse.equalsIgnoreCase("Y")) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_YES);
  } else if (m_AutoResponse.equalsIgnoreCase("N")) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_NO);
  } else {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_DEF);
  } // end of else
}
```

図 24: メソッド引き上げに適したコードクローン (Ant)

承していた．これはメソッド引き上げ (2.1.2 参照) のリファクタリングが行える典型的な例である．そのメソッドのソースコードを図 24 に示す．

また抽出を行わなかった場合と比較し，クローンペア数は $\frac{1}{120}$ に，クローンクラス数は $\frac{1}{16}$ になっている．

5.3 メトリクスを用いたコードクローンの抽出の適用実験

5.3.1 概要

前節の適用実験においては，コンパウンド・ブロックの抽出により，メソッド抽出やメソッド引き上げのリファクタリングを容易に行うことができる例を一つずつ見つけることができた．本節では，メトリクスを用いたコードクローンの抽出を行うことで，それらと比べどのようなコードクローンを抽出すつことができるのか，またはリファクタリングに適したコードクローンを抽出することが可能であるかどうかを確認する．

本適用実験においては，Ant(5.2 参照) を対象として本手法の適用を行った．

5.3.2 分析

本手法の有効性を確認するため，以下の項目について分析を行った．

(1) 各メトリクス値とコードクローンの特性：各メトリクス間での相関を調べ，コードクローンの特性を調べる．そして各メトリクスについて，実際にどのようなクローンクラスが上位を占めているのか，実際にソースコードを参照することで各メトリクスの有意性を確認する．

(2) RFT 値の上位を占めるクローンクラスが本当にリファクタリングは可能なものであるのかどうか，またリファクタリングを行ったならばその効果はどの程度あるのかについて考察を行う．

(3) 抽出によるクローン散布図の変化：メトリクスによる抽出によりクローン散布図がどの程度変化することを見ることで，散布図上におけるメトリクスを用いた抽出の有効性を確認する．

本適用実験においては，CCFinder における最小一致トークン数は 30 トークンに定め，2,842 個のクローンクラス，19,657 組のクローンペアが検出された．以降，この検出結果に対して分析を行う．

(1) 各メトリクス値とコードクローンの特性

検出された 2,842 個の全クローンクラスについて 3.3.4 節において定義した，9 つのメトリクス，RAD，LEN，POP，DFL，LNR，RNR，CMP，USR，RFT の値を計測した．

表 7 に．各メトリクス値の間での順位相関係数を示す．

これらの中で，強い相関関係が見られたのは，LNR と RFT，CMP と RFT の間であった．これらの値の間に強い正の相関が見られるのは定義どおりであり，LEN と DFL，POP と DFL の間も同様のことが言える．

その他に，定義としては全く異なるが，LNR と CMP の間にもやや強い正の相関が見られた．これは，制御構造が複雑なものほど，繰り返されにくく，コードサイズが大きくなるという直観に当てはまっている．

また，逆に，USR と RFT の間には，意図した定義とは異なる相関が見られた．定義上，基本的に USR が大きくなればなるほど，RFT は大きくなるはずであるがほとんど相関がなかった．これは，USR と RNR の相関で説明できる．これらの間には負の相関がややみられ，ユーザ定義名の種類数が増えれば増えるほど，繰り返されやすくなっていたと考えられる．これも一見矛盾したように思えるが，変数宣言部に多くのコードクローンが検出され

表 7: 各メトリクス値の間での順位相関係数 (0.4 以上を強調表示)

	RAD	LEN	POP	DFL	LNR	RNR	CMP	USR	RFT
RAD	1.000								
LEN	-0.352	1.000							
POP	**0.462**	-0.210	1.000						
DFL	0.115	**0.499**	**0.676**	1.000					
LNR	-0.194	**0.457**	-0.131	0.168	1.000				
RNR	0.075	-0.226	0.041	-0.173	**0.606**	1.000			
CMP	-0.020	0.170	-0.009	0.106	**0.692**	**0.572**	1.000		
USR	0.063	**0.400**	-0.020	0.278	0.039	-0.333	-0.149	1.000	
RFT	0.089	0.193	0.235	0.317	**0.745**	0.614	**0.814**	0.014	1.000

```
1|Enumeration e = pathComponents.elements();
2|while (e.hasMoreElements()){
3|    File pathComponent = (File) e.nextElement();
      :
```

図 25: (分類 2) $Enumeration$ インタフェースを介したループ定型処理コード

た場合などは，名前の違う変数が多く現れ USR 値が大きくなる，しかも変数宣言部であるので繰り返し部分が多く，非繰り返しコードの割合 RNR は減少する．RNR が減少すると，RFT の定義上，0 へ切捨てられる可能性がある．つまり逆に言えば，USR と RFT の相関が 0.014 と小さかったのは，変数宣言部のコードクローンが多く検出されたことを証明している．

そして，RAD と POP の間のやや強い正の相関が見られるのは，要素数が多くなればなるほど，いろんなところに分散している可能性が高くなるため直観的な理解に反さない．RAD と LEN の間に少し負の相関が見られるのも，離れれば離れるほど，サブシステムは異なった機能を持つはずであり，遠く離れた箇所に大きなコードクローンが存在する可能性が低いことは納得がいく．逆に，仮にこの値が正の値に近づくと，遠く離れたサブシステムが類似した機能を持っているかもしれず，設計上 (ファイルシステムの距離を設計構造の距離と考えるならば) 問題がある可能性があると言える．

次に，これらのメトリクスのうち，LEN，POP，DFL，LNR，CMP，USR，RFT の 7 つそれぞれに関して，上位 10 位までのクローンクラスを調べたところ次の 6 つに分類することができた (各メトリクス上位 10 位までのクローンクラスの全メトリクス値はそれぞれ付録，表 9 から表 15 参照)．

分類 **1** 変数宣言のコード

> フィールド変数，もしくメソッド内ローカル変数の連続した宣言文におけるコードクローン．自己交差の関係にあるコードクローンも含まれる．自己交差の関係とは，クローンペアの関係にある 2 つのコード片について，ともに同一ファイル内に存在し，前方に位置するコード片の終了位置が，他方のコード片の開始位置より後方に位置する場合，そのクローンペアは自己交差の関係にあるという．

分類 **2** $Enumeration$ インタフェースを介したループ定型処理コード (図 25 参照)

> $Enumeration$ インターフェースとは，ベクタの要素や，ハッシュテーブルのキー，および値の列挙子である．ループ文を用いてベクタ等のデータを順に参照していく定型処理コードの部分が一致したクローンクラス．

分類 **3** メソッド内での繰り返しコード

```
  1|public void init() {
  2|    String p;
  3|
  4|    if ((p = project.getProperty("ant.netrexxc.binary"))  != null) {
  5|        this.binary = Project.toBoolean(p);
  6|    }
  7|    // classpath makes no sense
  8|    if ((p = project.getProperty("ant.netrexxc.comments"))  != null) {
  9|        this.comments = Project.toBoolean(p);
 10|    }
 11|    if ((p = project.getProperty("ant.netrexxc.compact"))  != null) {
 12|        this.compact = Project.toBoolean(p);
 13|    }
           :
           :
100|    if ((p = project.getProperty("ant.netrexxc.suppressDeprecation"))  != null) {
101|        this.suppressDeprecation = Project.toBoolean(p);
102|    }
103|}
```

図 26: (分類 3.1) 簡単な if ブロックの繰り返しコード

3.1 簡単な if ブロックの繰り返しコード (図 26 参照)

簡単な if ブロックが何度も繰り返されており，これらの繰り返しコード内での自己交差の関係にあるクローンペアで構成されるクローンクラス．

3.2 パフォーマンスの向上を意図した完全一致コードの繰り返しコード (図 27 参照)

図 27 のソースコード中には完全に一致したコード (図中で (#) の部分) が 3 回繰り返されており，ソースコードのコメントの中には，明示的に “/* copy 2 */”, “/* copy 3 */” と記述されていた．このコードクローンが存在したメソッド (simpleSort) が呼び出されている部分を参照したところ，本メソッドは扱うデータサイズが小さい場合のソートのために呼ばれており，その呼び出しコードの直前には

“/* Use simpleSort(), since the full sorting mechanism
 has quite a large constant overhead. */”

というコメントが記述されていた．つまり，扱うデータサイズが小さい時には，ソートに必要なオーバーヘッドを排除したパフォーマンスの向上のための意図的なコードクローンであることが確認された．また実際，一般的にパフォーマンスが重要となる，データの圧縮処理を行うクラスであった．

3.3 繰り返し処理の 1 度目と 2 度目以降の処理の扱いが異なるコード (図 28 参照)

基本的には同じような処理の繰り返しであるが，初回のみ扱いが異なるため重複したロジックが複数箇所 (ループの内側と外側) に存在していた．

```
 1|private void simpleSort(int lo, int hi, int d) {
        :
20|        while (true) {
21|            /* copy 1 */
                //(#) begin
22|            if (i > hi) {
23|                break;
24|            }
25|            v = zptr[i];
26|            j = i;
27|            while (fullGtU(zptr[j - h] + d, v + d)) {
28|                zptr[j] = zptr[j - h];
29|                j = j - h;
30|                if (j <= (lo + h - 1)) {
31|                    break;
32|                }
33|            }
34|            zptr[j] = v;
35|            i++;
                //(#) end
36|
37|            /* copy 2 */
                // copy from (#) begin to (#) end
52|
53|            /* copy 3 */
                // copy from (#) begin to (#) end
68|
69|            if (workDone > workLimit && firstAttempt) {
70|                return;
71|            }
72|        }
        :
74|}
```

図 27: (分類 3.2) パフォーマンスの向上を意図した完全一致コードの繰り返しコード

分類 4 同一名のメソッド間の一致コード

同一名の複数メソッドの間で完全，もしくはかなりの一致が見られたクローンクラス．関連したメソッドは次の 5 つの名前のメソッドであった．

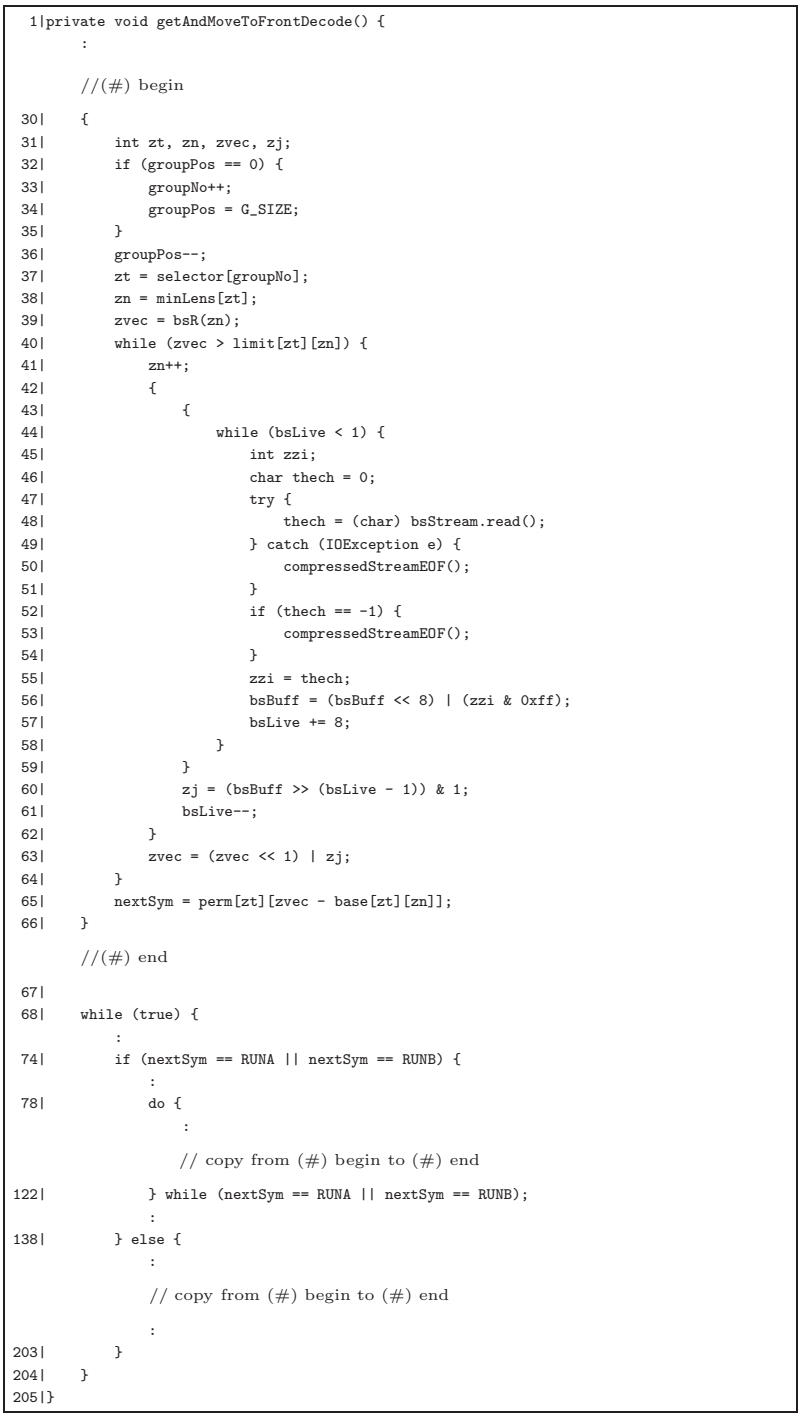
4.1 “isRebuildRequired”

4.2 “buildCmdLine”

4.3 “visit”

4.4 “exec”

4.5 “getCompatibilityWith”



```
 1|private void getAndMoveToFrontDecode() {
       :

       //(#) begin

30|    {
31|        int zt, zn, zvec, zj;
32|        if (groupPos == 0) {
33|            groupNo++;
34|            groupPos = G_SIZE;
35|        }
36|        groupPos--;
37|        zt = selector[groupNo];
38|        zn = minLens[zt];
39|        zvec = bsR(zn);
40|        while (zvec > limit[zt][zn]) {
41|            zn++;
42|            {
43|                {
44|                    while (bsLive < 1) {
45|                        int zzi;
46|                        char thech = 0;
47|                        try {
48|                            thech = (char) bsStream.read();
49|                        } catch (IOException e) {
50|                            compressedStreamEOF();
51|                        }
52|                        if (thech == -1) {
53|                            compressedStreamEOF();
54|                        }
55|                        zzi = thech;
56|                        bsBuff = (bsBuff << 8) | (zzi & 0xff);
57|                        bsLive += 8;
58|                    }
59|                }
60|                zj = (bsBuff >> (bsLive - 1)) & 1;
61|                bsLive--;
62|            }
63|            zvec = (zvec << 1) | zj;
64|        }
65|        nextSym = perm[zt][zvec - base[zt][zn]];
66|    }

       //(#) end

67|
68|    while (true) {
           :
74|        if (nextSym == RUNA || nextSym == RUNB) {
               :
78|            do {
                   :

                   // copy from (#) begin to (#) end

122|            } while (nextSym == RUNA || nextSym == RUNB);
               :
138|        } else {
               :

               // copy from (#) begin to (#) end

               :
203|        }
204|    }
205|}
```

図 28: (分類 3.3) 繰り返し処理の 1 度目と 2 度目以降の処理の扱いが異なるコード

分類 5 その他のメソッド間での一致コード

これらの分類を用いて，各メトリクス値の上位 10 位までのクローンクラスの分類を表 8

に示す．

まず，POP の上位を占めるのは，ほぼ分類 1 であり，USR と RFT の順位相関係数が小さかったことから分かったように，やはり変数宣言のコードクローンが多く検出されていたことが確認できる．それに伴い，DFL も上位には分類 1 のクローンクラスしか現れてこなかった．変数宣言においては，ひとつひとつのコードクローン片はさほど大きくなくとも，その構文の単純さ故に，他のクローンクラスに比べ POP 値が圧倒的に高く，DFL 値が高くなっている．また，同様に USR についても上位がほぼ分類 1 で占められている．これは，変数宣言において，名前の違う変数が多数現れることで USR 値が大きくなったクローンクラスが上位に現れている．

一方，LEN においては，大部分を分類 3.1 が占めている．図 26 の中のメソッドでは，多数あるプロパティ値を初期化する処理を行う．本メソッドにおいては，変数名と文字列定数が異なるだけの同じ `if` ブロックが 32 回も繰り返されていた．基本的に繰り返しコードは，配列やベクタを用いたループ化を行えばリファクタリングできる場合がある．しかし，本メソッドの直前には，

```
/* Sorry for the formatting, but that way it's easier to keep in sync
 with the private properties (line by line). */
```

というコメントが記述されていた．つまり，このメソッドの作成者は，このコードが書式としてはよくないということを認識しているが，このように書いたほうが利便性が高いため，意図的にこのようになっていることが確認された．また，`if` という予約語が多数存在するので CMP の上位も分類 3.1 で占められていることが確認できる．しかし，この作成者のように利便性が高いという見方もあるが，将来，新しいプロパティ設定機能を追加する場合などは，コピーとペーストと変数名等の書き換えで拡張が行われると予想される．その際には，部分的な変数名の書き換え忘れによる，単純ではあるが，繰り返しの中であるため視

表 8: 各メトリクス値の上位 10 位までのクローンクラスの分類

順位	LEN	POP	DFL	LNR	CMP	USR	RFT
1 位	3.1	1	1	4.2	3.1	1	4.2
2 位	3.1	1	1	4.1	3.1	1	4.1
3 位	3.1	1	1	4.1	3.1	4.2	4.2
4 位	3.1	1	1	4.2	3.1	1	4.1
5 位	3.1	1	1	4.2	3.1	1	4.1
6 位	3.1	1	1	4.3	3.1	1	4.3
7 位	3.1	1	1	4.1	3.1	1	4.1
8 位	4.1	1	1	3.2	3.1	1	4.5
9 位	4.2	2	1	3.3	3.1	1	5
10 位	4.1	1	1	4.4	3.1	1	2

覚的には発見し辛いバグの混入の危険性をはらんでおり，保守性の高いコードであるとは言い難い．

それらに対し，LNR，もしくは RFT の上位には分類 1 や分類 3.1 のような単純な繰り返しコードは一切含まれていない．特に，LNR，RFT ともに，主に，分類 4 や 5 のメソッドレベルでの類似が見られる箇所に存在するコードクローンが上位を占めている．これらの分類のクローンクラスについてのリファクタリングを行えるのかどうかに関しては後述するが，$2.1.2$ 節でのリファクタリング手法の多くがメソッド単位の手法であったように，一般に，メソッドレベルでの類似が見られれば，リファクタリングが行える可能性は高いと考えられる．したがって，他のメトリクス値に比べこの 2 つのメトリクスは，かなり有意義なコードクローンを抽出できていると言える．

しかし，上位 7 位までについては共に同じように分類 4.1，4.2，4.3 で占められているように，そもそも RFT 値は LNR 値を基に算出されているため，RFT に，LNR に対する有意性がないのかということに注意しなくてはならない．そこで，LNR と RFT，それぞれの上位 100 クローンクラスの中での，順位相関を調べた．その結果，それぞれの順位相関係数は，約 0.2 から 0.3 であり，それぞれの上位のクローンクラスには必ずしも相関があるとは言えなかった．明確なこの違いが見られるのは，特に RFT 値 10 位に現われている分類 2 であった．分類 2 は，POP 値 9 位に現れているように，図 25 のようなデータ構造アクセスに対する定型処理コードである．この定型処理を部品化することは一般的に意味を持たないかもしれないが，仮に部品化できるような定型処理コードが存在したならば，RFT の上位に現れてくる可能性があることを示している．また，逆に RFT 値が高いもの (100 位以内) の中で，POP 値も比較的高い (10 以上) クローンクラスを調べたところ，$Enumeration$ に対するループの定型処理以外に，図 29 のような `finally` ブロックに含まれた入出力ストリームに対するクローズ定型処理が見つかった．このような場合，アスペクト指向プログラミング [32] によるアプローチでコードクローンを排除できるかもしれない．

基本的に LNR の値を用いることにより，メソッドレベルでの類似が見られるようなコードクローンが抽出されやすいことが確認されたが，その LNR に対し，CMP や USR が有意義なコードクローンを抽出することができないのかどうかを確認するため，LNR 値が低いが CMP や USR が高くなっているクローンクラスについて調べた．

まず，LNR 値が低いが CMP が高くなっているクローンクラスを調べるため，正規化された CMP を正規化された LNR で除した値を基に上位から参照を行ったところ，その大部分はやはり分類 3.1 のように `if` ブロックが連続したコードであった．定義からしてごく自然な結果ではあるが，変数宣言の連続等の無意味なものではなく，上述の通りバグの混入の危険性をはらんだ連続コードを抽出できる (同時に LNR が高いものの中からは抽出できるとは限らない) という意味で意義があると考えられる．

```
  1|protected void runCommand(Commandline toExecute) throws BuildException {
        :
 68|    try {
           :
 97|    } catch (Exception e) {
           :
103|    } finally {
104|        if (outputStream != null) {
105|            try {
106|                outputStream.close();
107|            } catch (IOException e) {}
108|        }
109|        if (errorStream != null) {
110|            try {
111|                errorStream.close();
112|            } catch (IOException e) {}
113|        }
114|    }
115|}
```

図 29: finally ブロックに含まれた定型処理

同様に，LNR が低いが USR が高くなっているクローンクラスを調べるため，正規化された USR を正規化された LNR で除した値を基に上位から参照を行ったところ，図 30 のような，GUI コンポーネントの設定処理コードの中にコーディングスタイルによって類似が見られたものが見つかった．図 30 は，コンポーネントのレイアウトを設定する処理コードの一部である．煩雑なパラメータを，様々なコンポーネントに対し同じようなコーディングスタイルで設定処理を行っている．このようなデータの煩雑さの密度が比較的高いもの発見できる (単なる LNR 値の大きさでは発見が難しい) 可能性は，意義があると考えられる．しかし，通常は，分類 1 のような変数宣言の連続コード等が LNR は低いが USR が高くなる．そこでここでは，変数宣言の連続のクローンクラスを除外する意味で，RAD を 0 に絞りこむことでファイル内で閉じているクローンクラスの参照を行った．それは，変数宣言等が一致したクローンクラスがファイル内で閉じている可能性が低いためである．

また 5.2.2 節にて機能的なまとまりのあるコードクローンとして抽出されたメソッド getAutoresponse(図 24 参照) は，RFT 値で 22 位と比較的上位に現れていたことを確認した．

(2) リファクタリング

RFT 値の上位を占めたクローンクラスが本当にリファクタリング可能なものであるのかどうか，またリファクタリングを行ったならばその効果はどの程度あるのかを示すため，LNR

```
 1|private Panel getOptionenPanel() {
                :
27|             GridBagConstraints constraintsProjectText = new GridBagConstraints();
28|             constraintsProjectText.gridx = 1; constraintsProjectText.gridy = 0;
29|             constraintsProjectText.gridwidth = 2;
30|             constraintsProjectText.fill = GridBagConstraints.HORIZONTAL;
31|             constraintsProjectText.anchor = GridBagConstraints.WEST;
32|             constraintsProjectText.insets = new Insets(4, 4, 4, 4);
33|             getOptionenPanel().add(getProjectText(), constraintsProjectText);
34|
35|             GridBagConstraints constraintsBuildFileTextField = new GridBagConstraints();
36|             constraintsBuildFileTextField.gridx = 1; constraintsBuildFileTextField.gridy = 1;
37|             constraintsBuildFileTextField.fill = GridBagConstraints.HORIZONTAL;
38|             constraintsBuildFileTextField.anchor = GridBagConstraints.WEST;
39|             constraintsBuildFileTextField.weightx = 1.0;
40|             constraintsBuildFileTextField.insets = new Insets(4, 4, 4, 4);
41|             getOptionenPanel().add(getBuildFileTextField(), constraintsBuildFileTextField);
                :
75|}
```

図 30: GUI コンポーネントの設定処理コード

と共に上位を占めていた，分類 4.1，4.2 の 2 つのメソッドについてリファクタリング例を示す．

(分類 **4.1**)**“isRebuildRequired”** メソッドのリファクタリング

isRebuildRequired メソッドは，約 1000 トークン (約 180 行) に及ぶ，比較的大きなメソッドである．本メソッドは，2 つのクラス WeblogicDeploymentTool，WebsphereDeploymentTool の中で protected メソッドとして定義されていた．また，それら 2 つのクラスは共に同じスーパークラス GenericDeploymentTool を継承していたため，基本的に本メソッドをスーパークラスへ引き上げることを考える．しかし，クラスの継承関係等を図 31 の上部に示すが，GenericDeploymentTool クラスを継承していたクラスはその 2 クラスだけでなく，WeblogicDeploymentTool，WebsphereDeploymentTool の他に 4 クラス存在し，その 4 クラスの中では，本メソッドは定義されていなかった．よって，GenericDeploymentTool クラスの汎用性を保つため，GenericDeploymentTool クラスを継承した WebDeploymentTool クラスを新たに定義し，WeblogicDeploymentTool，WebsphereDeploymentTool では WebDeploymentTool を継承するように変更することを考える．

したがって，WebDeploymentTool クラスへ本メソッドを引き上げることができればよい．しかし，本メソッドが定義されていた 2 つのクラスの間では，図 32 で示すように，

相違点 **1** “(#1)” の箇所の break 文の実行位置，

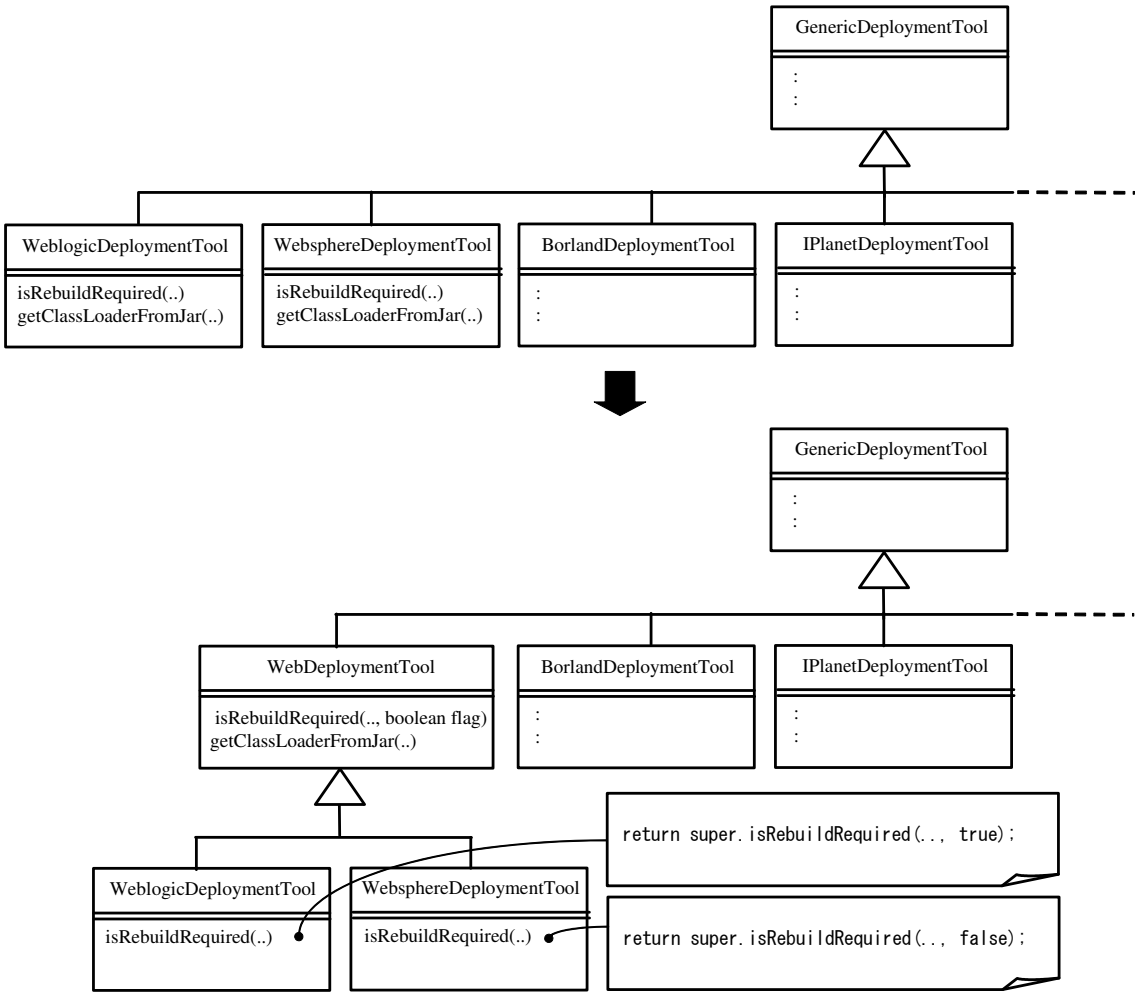


図 31: (分類 4.1)“isRebuildRequired” メソッドのリファクタリング例

相違点 **2** “(#2)” の `if` ブロックの有無

の 2 点で若干実装が異なっていた . この状況に対する , 本メソッドの引き上げ方法は当然様々あるが , 単純にフラグを用いてその違いを吸収することにする . つまり , 引き上げるメソッドの引数にフラグ引数を追加して , その値によって , 相違点 1 , 2 の実行の切り替えを行えばよい . そしてサブクラス `WeblogicDeploymentTool` , `WebsphereDeploymentTool` の `WisRebuildRequired` で , フラグ引数を伴ったスーパークラスの `WisRebuildRequired` の呼び出しを行うようオーバーライドする .

さらに , メソッド引き上げの際 , 同時に考えなくてはならないことは , フィールド変数の引き上げであるが , 本メソッドにおいてはフィールド変数の参照 , 代入はなかった . しかし , `GenericDeploymentTool` で定義されていないメソッド

```
 1|// "WebsphereDeploymentTool"クラス ベース
 2|protected boolean isRebuildRequired(File genericJarFile, File websphereJarFile){
 3| try {
 4|     :
 5|     if (genericJarFile.exists() && genericJarFile.isFile()
 6|          && websphereJarFile.exists() && websphereJarFile.isFile()){
 7|         :
 8|         for (Enumeration e = genericEntries.keys(); e.hasMoreElements();){
 9|             :
10|             if (wasEntries.containsKey(filepath)){
11|                 :
12|                 if ((genericEntry.getCrc() != wasEntry.getCrc()) ||
13|                     (genericEntry.getSize() != wasEntry.getSize())){
14|                     if (genericEntry.getName().endsWith(".class")){
15|                         :
16|                     }else{
17|                         if (!genericEntry.getName().equals("META-INF/MANIFEST.MF")){
18|                             :
19|                             // "WeblogicDeploymentTool"クラスでは (#1) の代わりにここで"break;"
20|                         }
21|                         break;// (#1)
22|                     }
23|                 }
24|             }else{
25|                 :
26|             }
27|         }
28|         :
29|
30|         // (#2) は"WeblogicDeploymentTool"クラスのみで記述
31|         // (#2) begin
32|         if (genericLoader instanceof AntClassLoader){
33|             AntClassLoader loader = (AntClassLoader)genericLoader;
34|             loader.cleanup();
35|         }
36|         // (#2) end
37|     }
38| }
39| :
40|}
```

図 32: (分類 4.1)“isRebuildRequired” メソッドにおける不一致部分 (“WebsphereDeploymentTool” クラス側でのコードをベース)

getClassLoaderFromJar の呼び出しが含まれていた．このメソッドは WeblogicDeploymentTool ,WebsphereDeploymentTool 両方のクラスで定義されており，しかもそれらは文字列レベルで完全一致していた．したがって，isRebuildRequired メソッドの引き上げと同時に，getClassLoaderFromJar をそのままの形で WebDeploymentTool に引き上げることでリファクタリングは完了する．そのリファクタリングの結果，図 31 の下部のような継承関係になる．

本メソッドのリファクタリングの効果は，行数で言えば約 100 行から 150 行程度のコードサイズ減少につながる．実際にソースコードを変更し，コンパイル，実行した訳ではないので，その他の定量的，客観的な評価は難しいが，フィールド変数の引き上げ等も必要なく容易に引き上げることができ，同時に getClassLoaderFromJar メソッドを引き上げることができたことでその効果は決して低いものでないと考えられる．

但し，フラグを使ったことによる，コードの理解性の低下が問題になるため，今後の拡張が予測できる際には，よりよい形での引き上げを行う必要があるかもしれない．

(分類 4.2) **“buildCmdLine” メソッドのリファクタリング**

buildCmdLine メソッドは，約 500 トークン (約 80 行) のメソッドである．本メソッドは，4 つのクラス SOSCheckin，SOSCheckout，SOSGet，SOSLabel，の中で protected メソッドとして定義されていた．また，isRebuildRequired メソッドと同様，それら 4 つのクラスは共に同じスーパークラス SOS を継承していており，逆に，SOS を継承していたクラスはこの 4 つのみであった．そこで本メソッドを SOS へ引き上げることを考える．これらのクラスの継承関係等を図 33 の上部に示す．

しかし，これらの 4 つのクラスにおける buildCmdLine メソッドは完全に一致していたわけではなく，互いに少しずつ実装されている機能が異なっていた．4 つのクラスそれぞれにおける buildCmdLine メソッドで実装されていた機能を，図 34 に示す．

図 34 の表中では，各列がそれぞれのクラス内での buildCmdLine メソッドを表し，ソースコード列上での順序と同じ順でコード片毎の機能を表す f1，f2，... が上から並んでいる．これらのコード片毎の機能は，ある程度独立性の高い機能となっており，それらの組合せで各メソッドは実装されていた．さらに，各コード片は，機能が同じであれば文字列レベルでの一致が見られた．但し，機能 f2 については機能は似通っているものの，各クラス毎に若干異なるため，f2’，f2’’，f2’’’ となっている．

本メソッドを SOS へ引き上げることをこの表から考えると，全 4 クラスで実装されている部分 f1，f3，...，f8 については，引き上げ後のメソッドの中で実装されればよく，それに対し，全てのクラスで実装されているわけではない機能 f8，...，f12，およびクラス毎に異なる機能 f2，...，f2’’’ については，それらの機能選択をサブクラス側でそれぞれ実装できるようする必要がある．

この状況に対する本メソッドの引き上げ方法もやはり様々考えられるが，ここではテンプレートメソッドの形成を行う．つまり，引き上られたメソッドの中で機能選択が必要な部分を新たなメソッド M1，M2 の呼び出しとして置き換え，メソッド M1，M2 の実装はサブクラスで行うようにする (メソッド M1，M2 は本来抽象メソッドとして定義すべきだが，空メソッドのままにしておく)．そしてサブクラスの M1 では f2，...，f2’’’ のいずれかの機能へオーバーライドする．M2 では，f8，...，f12 のうち必要な機能へオーバーライドする．これらは，f8，...，f11 と f12 の 2 グループに分類することができるのでそれらをさらにスーパークラス SOS で新たなメソッド m1，m2 として定義する．

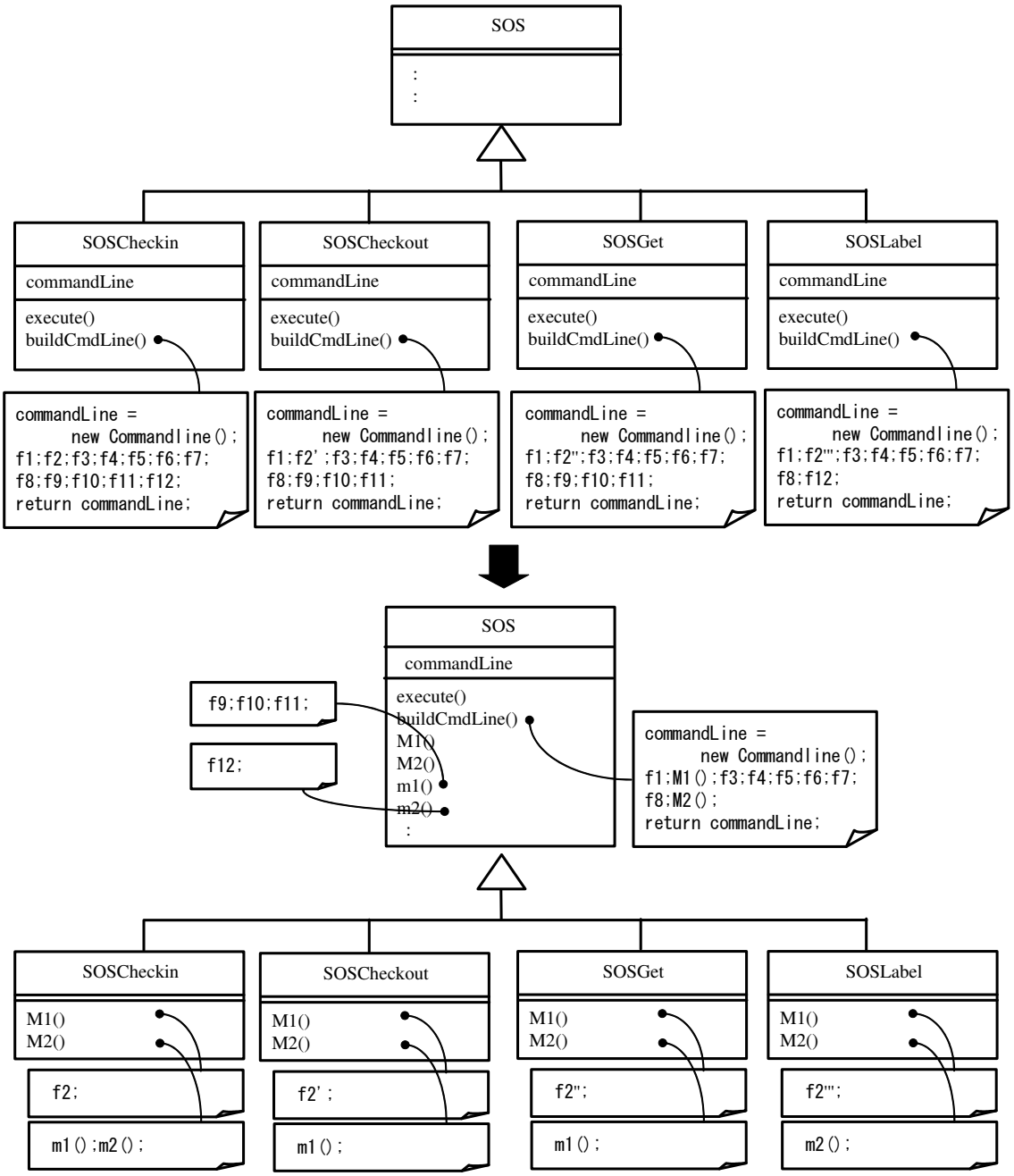


図 33: (分類 4.2)“buildCmdLine” メソッドのリファクタリング例

これらを用いてリファクタリングを行うと，図 33 の下部のようになる．本メソッドのリファクタリング分析に際し，完全に一致していた `execute` メソッドも容易に引き上げることが可能であることが分かったため，図中に示されている．

本メソッドのリファクタリングの効果は，行数で言えば約 200 行から 250 行程度のコー

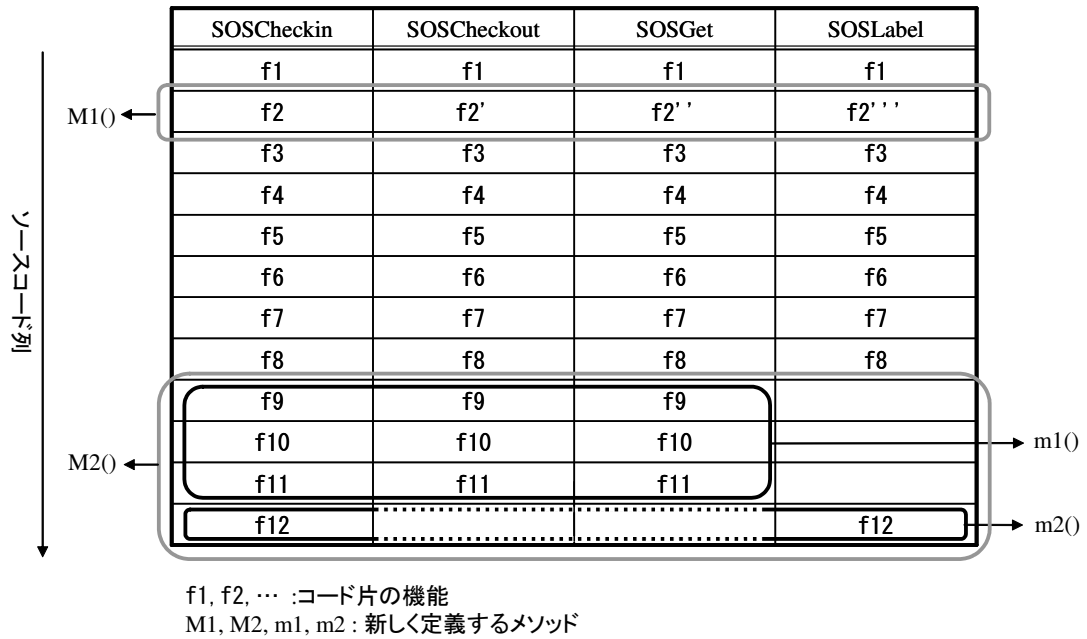


図 34: (分類 4.2) 各 “buildCmdLine” メソッドの実装機能内容とそのメソッド抽出

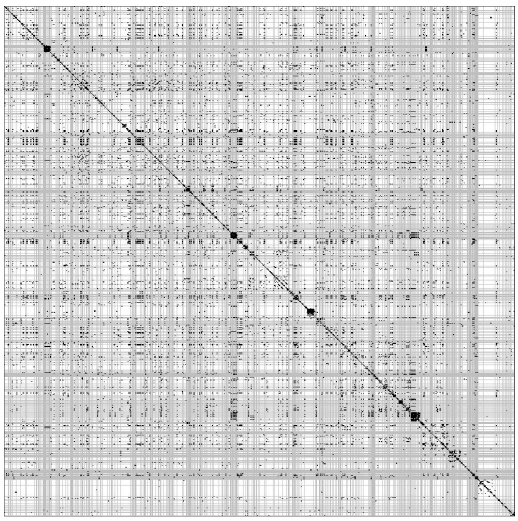
ドサイズ減少につながる．やはり実際にソースコードを変更し，コンパイル，実行した訳ではないので，その他の定量的，客観的な評価は難しいが，`execute` メソッドも同時に引き上げることで，各サブクラスは数行程度の非常に小さなクラスにすることができる．

(3) 抽出によるクローン散布図の変化

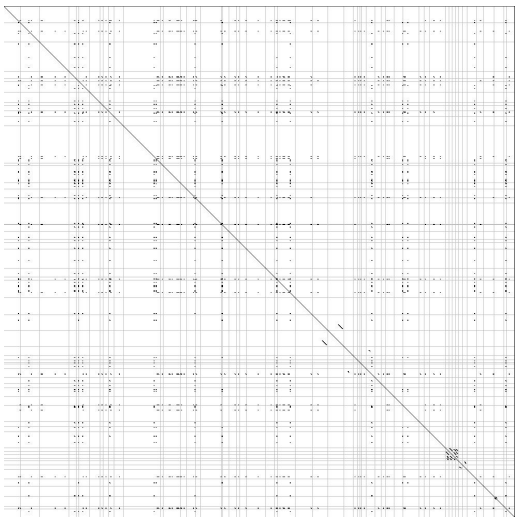
Gemini におけるメトリクスグラフを用いて実際にメトリクスを用いた抽出がクローン散布図上においてどのような効果があるのかを確認する．

まず，図 35(a) が，全く抽出を行わなかった場合のクローン散布図であり，ここには検出された全 2842 クローンクラスが表示され，それらに関連したファイル全 382 ファイルが座標軸上に配置されている．図中での格子は各ファイル間の区切り位置を示している．見ての通り，大量のコードクローンが存在しどこに着目すべきかという識別性は全くない．また，広く全体にコードクローンが分布しているため，類似度を用いて散布図座標軸に並ぶファイルの順番にソートをかけてもクローンペアが密集している箇所が多数存在し，ズーム機能を用いても密集箇所の確認作業に相当の手間がかかることが予想された．

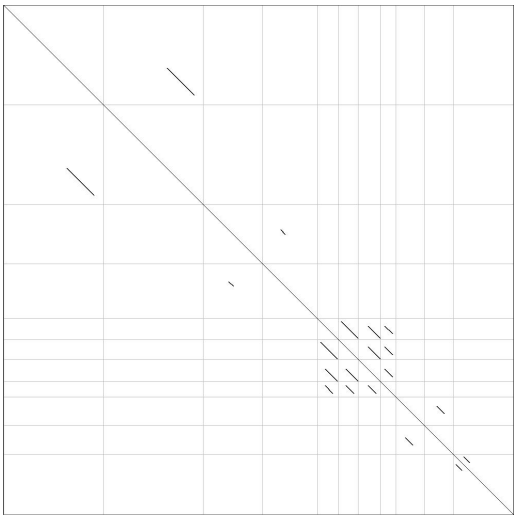
それに対し，図 35(b) が，RFT による抽出を行った場合である．抽出範囲は，RFT 値上



(a) 抽出を行わなかった場合 (2842 クローンクラス)



(b) 抽出を行った場合 (RFT 値上位 10 位までの 10 クローンクラス)



(c) 抽出を行った場合 (RFT 値上位 10 位までのクローンクラスに RAD 値 5 未満の絞り込みをかけた 9 クローンクラス)

図 35: クローン散布図 (Ant，最小一致トークン数 30)

位 10 位までの 10 クローンクラスである．上述の通り，これらの 10 クローンクラスの中にはファイルシステム内に広範に広がっていた分類 2 のクローンクラスが含まれているため，関連ファイル数としては 57 ファイル存在し，しかし図中に表示されているのはメトリクス

により抽出されたクローンクラス内に含まれるクローンペアのみであるため何らかの特徴を備えたものである可能性が高く，変数宣言の連続コード等の無駄なコードクローンを参照してしまうといった無駄な確認作業は非常に少ない．

さらに，図 35(c) は，RAD 値による絞り込みを行うことで上位 10 位までの 10 クローンクラスのうち，RAD の低い 9 クローンクラスを抽出した場合である．それらに関連したファイルはわずか 11 ファイルであり，わざわざズーム機能を用いることもなく，一目でどのファイルとどのファイルに特徴あるコードクローンが存在しているのか確認することができる．

このように，様々なメトリクス値の絞りこみを組み合わせることで，特徴あるクローンクラスを容易に確認・参照できる．システムのスケーラビリティは実装に依存するが，コードクローン確認作業そのもののスケーラビリティを大幅に向上させるものであることが確認できた．

6 むすび

本研究では，これまでの CCFinder の様々なソフトウェアに対する適用の中から得られた評価を基に，コードクローン情報を用いた開発保守支援を目指した．その支援のため，主に不一致部分を含んだコードクローン (Gapped クローン) の検出手法の提案，および開発保守支援への利用に必要なコードクローンの抽出手法を提案した．利用に求められる抽出としては，機能的なまとまりをもったコードクローンの抽出と，メトリクスを用いたコードクローンの抽出を行った．

また，本手法を，支援環境 Gemini に実装し，いくつかのソフトウェアを対象にして適用実験を行い，その有効性を確認した．Gapped クローン検出手法においては，CCFinder のみでは発見が困難であったコピーとペーストとその修正によって生成されたコードクローンを発見することができた．機能的なまとまりをもったコードクローンの抽出と，メトリクスを用いたコードクローンの抽出においては，実際にリファクタリングを行えるようなコードクローンが容易に抽出可能であることが確認され，さらにそのリファクタリング例を示した．またこれらの抽出によりコードクローン確認作業のスケーラビリティが向上するものと予想された．

最後に今後の課題としては，本システムにより抽出されたコードクローンに対し実際にリファクタリングを行い，リファクタリングを行ったことによってどの程度プログラムの変更容易性が向上しているのかその効果の定量的，客観的な評価を得る必要がある．また，Gapped クローン検出機能，機能的にまとまりのあるコードクローンの抽出機能，メトリクスを用いたコードクローンの抽出手法の 3 つの機能は実装上の連携がとれていないため，その実装の改善およびそれらを連携し組み合わせた分析等が必要になる．そして，実際のソフトウェアの開発保守作業への適用や，さらに大規模なソフトウェアへの適用が考えられる．

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 科学技術振興事業団 さきがけ研究 21 神谷 年洋 研究員に深く感謝致します．

本研究において，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 肥後 芳樹 氏に深く感謝致します．

本研究において，御協力を頂きました 大阪大学 基礎工学部 情報科学科 泉田 聡介 氏に深く感謝致します．

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] A. J. Albrecht, "Measuring Application Development Productivity", *Proceedings the IBM Application Development Symposium*, 1979, 83-92.

[3] Ant, `http://jakarta.apache.org/ant/`, [Last visited 1st Feb. 2003].

[4] ANTLR, `http://www.antlr.org/`, [Last visited 1st Feb. 2003].

[5] G. Antoniol, G. Casazza, M.D. Penta, and E. Merlo, "Modeling Clones Evolution Through Time Series", *Proceedings IEEE International Conference on Software Maintenance-2001*, 2001, 273-280.

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

[7] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", *Proceedings the 2nd Working Conference on Reverse Engineering*, 1995, 86-95.

[8] B.S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance", *SIAM Journal on Computing*, 1997, 26(5):1343-1362.

[9] 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.

[10] 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.

[11] 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.

[12] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees", *Proceedings IEEE International Conference on Software Maintenance-1998*, 1998, 368-377.

[13] 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.

[14] N. Chapin, "A Measure of Software Complexity", *Proceedings the 1979 National Computer Conference*, 1979, 995-1002.

[15] E. T. Chen, "Program Complexity and Programmer Productivity", *IEEE Transactions on Software Engineering*, 1978, 4(3):187-194.

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

[17] S. Ducasse, M. Rieger, and S. Demeyer. "A Language Independent Approach for Detecting Duplicated Code", *Proceedings IEEE International Conference on Software Maintenance-1999*, 1999, 109-118.

[18] H. E. Dunsmore, and J. D. Gannon, "Data Referencing: An Empirical Investigation", *IEEE Computer*, 1979, 12(12):50-59.

[19] J. L. Elshoff, "An Analysis of Some Commercial PL/I Programs", *IEEE Transactions on Software Engineering*, 1976, 2(2):113-120.

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

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

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

[23] M.H. Halstead, *Elements of software science*, Elsevier North-Holland, 1977.

[24] J. Helfman, "Dotplot Patterns: a Literal Look at Pattern Languages", *TAPOS*, 1995, 2(1):31-41.

[25] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "On software maintenance process improvement based on code clone analysis", *Proceedings 4th International Conference on Product Focused Software Process Improvement*, 2002, 185-197.

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

[27] 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, 2001, 18(5):47-54.

[28] 泉田聡介, “ソフトウェア保守のためのコードクローン情報検索ツール”, 大阪大学基礎工学部情報科学科 特別研究報告, 2003.

[29] J.H. Johnson, “Identifying Redundancy in Source Code using Fingerprints”, *Proceedings CASCON'93*, 1993, 171-183.

[30] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code”, *IEEE Transactions on Software Engineering*, 2002, 28(7):654-670.

[31] 片平真史, 石浜直樹, “ソフトウェア独立検証及び評価 (IV&V) の研究”, `http://giken.tksc.nasda.go.jp/seika/gaiyou/H13/02files/` [Last visited 1 Feb. 2003].

[32] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin, “Aspect Oriented Programming”, *Proceedings European Conference on Object-Oriented Programming*, 1997, 1241:220-242.

[33] R. Komondoor, and S. Horwitz, “Using slicing to identify duplication in source code”, *Proceedings 8th International Symposium on Static Analysis*, 2001.

[34] J. Krinke, “Identifying Similar Code with Program Dependence Graphs”, *Proceedings 8th Working Conference on Reverse Engineering*, 2001, 562-584.

[35] B. Lagüe, D. Proulx, J. Mayrand, E. M. Merlo, J. Hudepohl, “Assessing the Benefits of Incorporating Function Clone Detection in a Development Process”, *Proceedings IEEE International Conference on Software Maintenance-1997*, 1997, 314-321.

[36] J. Mayland, C. Leblanc, and E. Merlo, “Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics”, *Proceedings IEEE International Conference on Software Maintenance-1996*, 1996, 244-253.

[37] T. J. McCabe, “A Complexity Measure”, *IEEE Transactions on Software Engineering*, 1976, 2(4):308-320.

[38] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, “Software Quality Analysis by Code Clones in Industrial Legacy Software”, *Proceedings 8th IEEE International Symposium on Software Metrics*, 2002, 87-94.

[39] M. Rieger, and S. Ducasse, “Visual Detection of Duplicated Code”, *Proceedings ECOOP’98 Workshop on Experiences in Object-Oriented Re-Engineering*, 1998, 75-76.

[40] L. Prechelt, G. Malpohl, and M. Philippsen, “Finding plagiarisms among a set of programs with JPlag”, *resubmitted to Journal of Universal Computer Science*, 2001. `http://www.ipd.uka.de/~prechelt/Biblio/#jplag` [Last visited 1 Feb. 2002].

[41] T.M. Pigoski, “Maintenance”, *Encyclopedia of Software Engineering*, 1994, 1:619-636.

[42] 植田泰士, 神谷年洋, 楠本真二, 井上克郎, “クローン検出ツールを用いたソースコード分析ツールの試作”, 電子情報通信学会技術研究報告 *SS2001-14*, 2001, 101(240):17-24.

[43] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “Gemini: Code Clone Analysis Tool”, *Proceedings 1st International Symposium on Empirical Software Engineering*, 2002, 2:31-32.

[44] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “Gemini: Maintenance Support Environment Based on Code Clone Analysis”, *Proceedings 8th IEEE International Symposium on Software Metrics*, 2002, 67-76.

[45] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “On Detection of Gapped Code Clones using Gap Locations”, *Proceedings 9th Asia Pacific Software Engineering Conference*, 2002, 327-336.

[46] M. R. Woodward, M. A. Hennell, and D. Hedley, “A measure to control flow complexity in program text”, *IEEE Transactions on Software Engineering*, 1979, 5(1):45-50.

[47] 山田茂，高橋宗雄，“ソフトウェアマネジメントモデル入門—ソフトウェアの品質の可視化と評価法”, 共立出版株式会社, 1993.

[48] 山本哲男, 松下誠, 神谷年洋, 井上克郎, “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, 2002, J85-D-I(6):503-511.

[49] S.W.L. Yip, and T. Lam, “A Software Maintenance Survey”, *Procceedings 1st Asia-Pacific Software Engineering Conference*, 1994, 70-79.

[50] J. C. Zolnowski, and D. B. Simmons, “Taking the Measure of Program Complexity”, *Proceedings National Computer Conference*, 1981, 329-336.

表 9: LEN 値上位 10 位までのクローンクラスの全メトリクス値 (および順位)

RAD	LEN	POP	DFL	LNR	RNR	CMP	USR	RFT
0(1928)	663(1)	2(1218)	1309(29)	51(305)	7(2793)	24(1)	30(34)	0(2207)
0(1928)	635(2)	2(1218)	1253(30)	51(305)	8(2776)	23(2)	29(39)	0(2207)
0(1928)	607(3)	2(1218)	1197(33)	51(305)	8(2776)	22(3)	28(43)	0(2207)
0(1928)	579(4)	2(1218)	1141(35)	51(305)	8(2776)	21(4)	27(48)	0(2207)
0(1928)	551(5)	2(1218)	1085(37)	51(305)	9(2755)	20(5)	26(53)	0(2207)
0(1928)	523(6)	2(1218)	1029(44)	51(305)	9(2755)	19(6)	25(57)	0(2207)
0(1928)	495(7)	2(1218)	973(46)	51(305)	10(2741)	18(7)	24(65)	0(2207)
1(1430)	487(8)	2(1218)	957(47)	387(2)	79(1776)	7(20)	34(25)	266604(2)
1(1430)	474(9)	2(1218)	931(49)	397(1)	83(1651)	9(16)	53(3)	540855(1)
0(1928)	467(10)	2(1218)	917(50)	51(305)	10(2741)	17(8)	23(69)	0(2207)

表 10: POP 値上位 10 位までのクローンクラスの全メトリクス値 (および順位)

RAD	LEN	POP	DFL	LNR	RNR	CMP	USR	RFT
4(55)	30(2626)	140(1)	4183(2)	7(2694)	23(2546)	0(1689)	12(566)	0(2207)
3(346)	31(2440)	137(2)	4230(1)	7(2694)	22(2564)	0(1689)	12(566)	0(2207)
3(346)	32(2284)	131(3)	4175(3)	7(2694)	21(2577)	0(1689)	13(405)	0(2207)
3(346)	33(2140)	114(4)	3745(4)	7(2694)	21(2577)	0(1689)	13(405)	0(2207)
3(346)	34(1992)	104(5)	3519(7)	7(2694)	20(2591)	0(1689)	13(405)	0(2207)
3(346)	35(1863)	102(6)	3553(6)	7(2694)	20(2591)	0(1689)	13(405)	0(2207)
3(346)	37(1611)	99(7)	3646(5)	7(2694)	18(2632)	0(1689)	15(259)	0(2207)
3(346)	38(1506)	88(8)	3327(8)	7(2694)	18(2632)	0(1689)	15(259)	0(2207)
5(2)	30(2626)	87(9)	2593(19)	27(1930)	90(1015)	1(698)	9(1254)	43490(10)
3(346)	42(1163)	79(10)	3301(9)	7(2694)	16(2658)	0(1689)	16(203)	0(2207)

表 11: DFL 値上位 10 位までのクローンクラスの全メトリクス値 (および順位)

RAD	LEN	POP	DFL	LNR	RNR	CMP	USR	RFT
3(346)	31(2440)	137(2)	4230(1)	7(2694)	22(2564)	0(1689)	12(566)	0(2207)
4(55)	30(2626)	140(1)	4183(2)	7(2694)	23(2546)	0(1689)	12(566)	0(2207)
3(346)	32(2284)	131(3)	4175(3)	7(2694)	21(2577)	0(1689)	13(405)	0(2207)
3(346)	33(2140)	114(4)	3745(4)	7(2694)	21(2577)	0(1689)	13(405)	0(2207)
3(346)	37(1611)	99(7)	3646(5)	7(2694)	18(2632)	0(1689)	15(259)	0(2207)
3(346)	35(1863)	102(6)	3553(6)	7(2694)	20(2591)	0(1689)	13(405)	0(2207)
3(346)	34(1992)	104(5)	3519(7)	7(2694)	20(2591)	0(1689)	13(405)	0(2207)
3(346)	38(1506)	88(8)	3327(8)	7(2694)	18(2632)	0(1689)	15(259)	0(2207)
3(346)	42(1163)	79(10)	3301(9)	7(2694)	16(2658)	0(1689)	16(203)	0(2207)
3(346)	43(1093)	72(11)	3079(10)	7(2694)	16(2658)	0(1689)	16(203)	0(2207)

表 12: LNR 値上位 10 位までのクローンクラスの全メトリクス値 (および順位)

RAD	LEN	POP	DFL	LNR	RNR	CMP	USR	RFT
1(1430)	474(9)	2(1218)	931(49)	397(1)	83(1651)	9(16)	53(3)	540855(1)
1(1430)	487(8)	2(1218)	957(47)	387(2)	79(1776)	7(20)	34(25)	266604(2)
1(1430)	356(14)	2(1218)	695(71)	298(3)	83(1651)	6(25)	24(65)	126106(4)
1(1430)	352(16)	2(1218)	687(73)	294(4)	83(1651)	6(25)	24(65)	124436(5)
1(1430)	309(18)	2(1218)	601(87)	282(5)	91(841)	6(25)	40(15)	197257(3)
1(1430)	214(29)	2(1218)	411(166)	202(6)	94(382)	5(34)	30(34)	89884(6)
1(1430)	224(26)	4(502)	879(52)	187(7)	83(1651)	4(51)	17(167)	77477(7)
0(1928)	179(51)	2(1218)	341(235)	167(8)	93(521)	7(20)	11(741)	38974(16)
0(1928)	179(51)	2(1218)	341(235)	162(9)	90(1015)	4(51)	20(107)	39422(13)
0(1928)	161(64)	2(1218)	305(291)	146(10)	90(1015)	4(51)	22(81)	39088(15)

表 13: CMP 値上位 10 位までのクローンクラスの全メトリクス値 (および順位)

RAD	LEN	POP	DFL	LNR	RNR	CMP	USR	RFT
0(1928)	663(1)	2(1218)	1309(29)	51(305)	7(2793)	24(1)	30(34)	0(2207)
0(1928)	635(2)	2(1218)	1253(30)	51(305)	8(2776)	23(2)	29(39)	0(2207)
0(1928)	607(3)	2(1218)	1197(33)	51(305)	8(2776)	22(3)	28(43)	0(2207)
0(1928)	579(4)	2(1218)	1141(35)	51(305)	8(2776)	21(4)	27(48)	0(2207)
0(1928)	551(5)	2(1218)	1085(37)	51(305)	9(2755)	20(5)	26(53)	0(2207)
0(1928)	523(6)	2(1218)	1029(44)	51(305)	9(2755)	19(6)	25(57)	0(2207)
0(1928)	495(7)	2(1218)	973(46)	51(305)	10(2741)	18(7)	24(65)	0(2207)
0(1928)	467(10)	2(1218)	917(50)	51(305)	10(2741)	17(8)	23(69)	0(2207)
0(1928)	439(11)	2(1218)	861(55)	51(305)	11(2726)	16(9)	22(81)	0(2207)
0(1928)	411(12)	2(1218)	805(61)	51(305)	12(2711)	15(10)	21(93)	0(2207)

表 14: USR 値上位 10 位までのクローンクラスの全メトリクス値 (および順位)

RAD	LEN	POP	DFL	LNR	RNR	CMP	USR	RFT
0(1928)	195(37)	2(1218)	373(192)	6(2778)	3(2832)	0(1689)	55(1)	0(2207)
0(1928)	190(40)	2(1218)	363(205)	6(2778)	3(2832)	0(1689)	54(2)	0(2207)
1(1430)	474(9)	2(1218)	931(49)	397(1)	83(1651)	9(16)	53(3)	540855(1)
0(1928)	185(45)	2(1218)	353(215)	6(2778)	3(2832)	0(1689)	53(3)	0(2207)
0(1928)	180(48)	2(1218)	343(223)	6(2778)	3(2832)	0(1689)	52(5)	0(2207)
0(1928)	175(54)	2(1218)	333(242)	6(2778)	3(2832)	0(1689)	51(6)	0(2207)
0(1928)	170(57)	2(1218)	323(258)	6(2778)	3(2832)	0(1689)	50(7)	0(2207)
0(1928)	165(61)	2(1218)	313(275)	6(2778)	3(2832)	0(1689)	49(8)	0(2207)
0(1928)	160(65)	2(1218)	303(293)	6(2778)	3(2832)	0(1689)	47(9)	0(2207)
0(1928)	155(70)	2(1218)	293(322)	6(2778)	3(2832)	0(1689)	46(10)	0(2207)

表 15: RFT 値上位 10 位までのクローンクラスの全メトリクス値 (および順位)

RAD	LEN	POP	DFL	LNR	RNR	CMP	USR	RFT
1(1430)	474(9)	2(1218)	931(49)	397(1)	83(1651)	9(16)	53(3)	540855(1)
1(1430)	487(8)	2(1218)	957(47)	387(2)	79(1776)	7(20)	34(25)	266604(2)
1(1430)	309(18)	2(1218)	601(87)	282(5)	91(841)	6(25)	40(15)	197257(3)
1(1430)	356(14)	2(1218)	695(71)	298(3)	83(1651)	6(25)	24(65)	126106(4)
1(1430)	352(16)	2(1218)	687(73)	294(4)	83(1651)	6(25)	24(65)	124436(5)
1(1430)	214(29)	2(1218)	411(166)	202(6)	94(382)	5(34)	30(34)	89884(6)
1(1430)	224(26)	4(502)	879(52)	187(7)	83(1651)	4(51)	17(167)	77477(7)
1(1430)	130(116)	2(1218)	243(455)	116(21)	89(1245)	7(20)	23(69)	55762(8)
0(1928)	171(56)	2(1218)	325(254)	144(11)	84(1607)	5(34)	21(93)	45550(9)
5(2)	30(2626)	87(9)	2593(19)	27(1930)	90(1015)	1(698)	9(1254)	43490(10)