独立行政法人情報処理推進機構 委託

2012 年度ソフトウェア工学分野の先導的研究支援事業
「コードクローン分析に基づくソフトウェア開発・保守支援に関する研究」
成果報告書

平成 25 年 1 月

国立大学法人大阪大学

本報告書は独立行政法人情報処理推進機構　技術本部　ソフトウェア・エンジニアリング・センターが実施した「2012 年度ソフトウェア工学分野の先導的研究支援事業」の公募による採択を受けて大阪大学大学院情報科学研究科（研究責任者　楠本真二）が実施した研究の成果をとりまとえたものである.

目次

研究成果概要 ····· 1

１．研究の背景および目的 ····· 11
1.1 背景 ····· 11
1.2. 研究課題 ····· 12
1.3 研究の意義 ····· 13

２．実施内容 ····· 14
2.1 研究アプローチ ····· 14
2.1.1 研究の全体像 ····· 14
2.1.2 関連するこれまでの研究について ····· 16
2.1.3 研究目標 ····· 29
2.2 研究の活動実績・経緯 ····· 30
2.3 研究実施体制 ····· 35

３．研究成果 ····· 38
3.1 研究目標 1「ソースコード理解支援」 ····· 38
3.1.1 当初の想定 ····· 38
3.1.2 研究プロセスと成果 ····· 38
3.1.3 実用化へ向けた課題と問題点 ····· 52
3.2 研究目標 2「リファクタリング支援」 ····· 52
3.2.1 当初の想定 ····· 52
3.2.2 研究プロセスと成果 ····· 53
3.2.3 実用化へ向けた課題と問題点 ····· 67
3.3 研究目標 3「再利用ライブラリ作成支援」,「違反流用コード発見支援」 ····· 67
3.3.1 当初の想定 ····· 67
3.3.2 研究プロセスと成果 ····· 68
3.3.3 実用化へ向けた課題と問題点 ····· 80

４．考察 ····· 81
4.1 研究により判明した効果や問題点等 ····· 81
4.1.1 ソースコード理解支援 ····· 81
4.1.2 リファクタリング支援 ····· 82
4.1.3 再利用ライブラリ作成支援・違反流用コード発見支援 ····· 83
4.2 今後の課題 ····· 85

参考文献 ····· 88

研究成果概要

１．背景

近年，ソフトウェアシステムは社会の至る所の基盤となっており，社会生活において必須のものになっている．また，我が国における情報サービス業の売上げは平成 23 年には約 19 兆円，その多くがソフトウェア開発関連業務によるものであり，経済界においても重要な産業分野と認識されている．一方，我が国におけるソフトウェア開発プロジェクトの成功率はあまり高くないという報告もある．最近でも，特許庁の情報システム開発の失敗により約 55 億円の損失が発生しており，ソフトウェア開発に対する技術面，管理面での更なる改善が求められている．

ソフトウェア開発,保守を阻害する問題の一つとしてこの 10 年間ソースコード上のコードクローンに関する問題が提起され，活発に研究がなされている．コードクローンとは，ソースコード上に存在する同一，または，類似したコード辺を意味する．ソースコードの流用により，バグの拡散，あるいは，機能追加時の一貫性確保の難しさ等の問題が指摘されている．特に，対象ソフトウェアが大規模な場合，チェックすべき箇所が膨大な数になってしまうこと，及び，人間がすべての重複部分を認識しておくことは難しいことから，大規模ソフトウェアからのコードクローン検出手法や分析結果の利用等が研究されてきている．ソフトウェア工学の分野で最も歴史と権威があるソフトウェア工学国際会議においてもコードクローンに関するセッションが企画されている．ソフトウェア保守に関する国際会議においても主要な研究テーマとなっている．更に，2007 年からはコードクローンに特化した国際会議も開催されている．

一方で，上述した国際会議等における，ソフトウェア開発企業からの報告というのはほとんどなされていない．すなわち，研究成果の開発現場への普及状況はあまり十分ではないと推察される．一部のソフトウェアツールベンダーがソースコード解析サービスとして展開しているものや，企業内でのローカルな利用にとどまっているのが現状である．その理由としては，一般的なコードクローン検出/分析手法を研究論文・研究発表の場で提示するだけでは不十分で，実際の開発現場における利用目的や状況に特化した手法の開発とその有用性の評価結果を合わせて提示することが現場への導入の高い動機付けになると考えられる．

２．目的

そこで本研究では，ソフトウェア開発や保守の様々な活動，状況（コンテキスト）に応じた支援を行う．具体的には，幾つかのコンテキストに応じたコードクローン検出手法の開発と検出されたコードクローンに対する対策手法の開発を行うことを目的とする．具体的には，以下の４つのテーマについて，支援手法の提案，プロトタイプの開発，有効性の評価を行った．

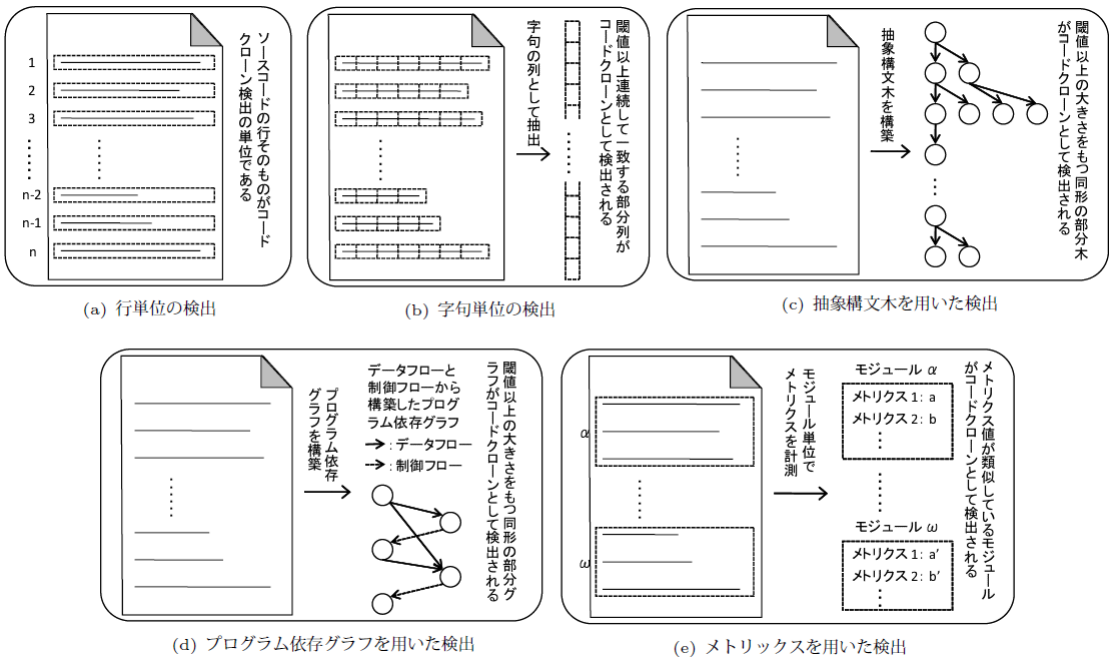
(G1) ソースコード理解支援
(G2) リファクタリング支援
(G3) 再利用ライブラリ作成支援
(G4) 違反流用コード発見支援

３．研究概要

(G1)～(G4)の研究を行う上で，大阪大学大学院情報科学研究科において，ソフトウェア工学，特に，ソフトウェア保守，プログラム解析，ソフトウェア信頼性等を専門としている７名の研究者で研究体制を構築した．具体的には，主に，(G1)を楠本真二，土屋達弘，岡野浩三の３名，(G2) を井上克郎，松下誠，石尾隆の３名，(G3) と(G4)を楠本真二，肥後芳樹の２名，が担当した．

４つのテーマを進めるにあたり，先ず，それぞれのテーマで対象とするコードクローン検出手法について検討した．コードクローン検出手法は，コードクローンをどの単位で検出するかによって，大きく以下の５つに分類できる（図 1)．

- 行単位：ソースコード上で閾値以上連続して一致する行の集まりをコードクローンとして検出する．すなわち，行そのものがコードクローン検出の単位となる．
- 字句単位：ソースコードを字句の列として抽出し，その上で閾値以上連続して一致する部分列をコードクローンとして検出する．
- 抽象構文木：ソースコードより抽象構文木を構築し，その上で閾値以上の大きさをもつ同形の部分木をコードクローンとして検出する．
- プログラム依存グラフ：ソースコードよりプログラム依存グラフを構築し，その上で閾値以上の大きさをもつ同形の部分木をコードクローンとして検出する．
- メトリクス:ソースコード中のモジュール毎にその特徴を表すメトリクス群を計測し，それらの値が類似しているモジュールをコードクローンとして検出する．



(a) 行単位の検出　(b) 字句単位の検出　(c) 抽象構文木を用いた検出

(d) プログラム依存グラフを用いた検出　(e) メトリックスを用いた検出

図１　コードクローン検出手法

直観的には，行単位，字句単位の検出手法は細粒度のコードクローンをなるべく多く高速に検出可能である．抽象構文木・プログラム依存グラフを用いた検出手法では，検出数

は少なくなるが意味的にまとまったものを検出可能である．しかし，グラフの構築・解析等のコストが高いため，大規模なソースコードに対してのスケーラビリティが低い．メトリクスを用いた検出手法は，計測するメトリクスにもよるが計測単位がモジュール単位でもあるので，高速な検出が可能である．但し，モジュール単位になるため計測の粒度は荒くなる．

以上の，検出手法の特徴を考慮して，(G1) では字句単位のコードクローン検出手法を，(G2) ではプログラム依存グラフを用いた検出手法を，(G3)，(G4) では，メトリクスを用いた検出手法を採用し，それぞれのテーマで，手法の開発，プロトタイプシステムの開発，評価実験を実施した．評価実験の実施に当たっては，既存手法と提案手法を，幾つかのオープンソースソフトウェアに対して適用し，結果の比較を行った．また，本研究では実用性を重視したため，実プロジェクトデータへの適用を試みた．具体的には，文部科学省「先導的 IT スペシャリスト育成推進プログラム」の一つである IT Spiral（関西圏 9 大学院が連携して，実践的ソフトウェア工学の教育を行う）で開発した実プロジェクト教材を用いた．これは，大学の教務情報管理システム（の一部）を実際のソフトウェア開発会社に発注し，開発した際に得られた現実の開発データを収集したものである．収集データの中にプログラムソースコード(Java で実装，規模は約 3 万行)が含まれており，それを利用した．

以降，(G1)～(G4)に関して，成果の概要をまとめる．

４．ソースコード理解支援

先ず，(G1) ソースコード理解支援であるが，ここでは，細粒度でなるべく多くのコードクローンを高速に検出する手法を開発した．具体的には，既存のコードクローン検出ツールと比較して，冗長なコードクローンをなるべく含まず，処理が高速で，検出したコードクローンの質が高い（再現率，適合率が高い）コードクローン検出ツールの開発を目指した．再現率は，「正解コードクローンの中の何割をツールが検出できるか」を表し，適合率は「ツールが検出したコードクローンの何割が正解であるか」を表す尺度である．なるべく多くのコードクローンを検出するという立場であれば，再現率が重視され，正しいコードクローンを検出するという立場であれば，適合率が重視される．細粒度でなるべく多くのコードクローンを検出できるツールとして著名なものに CCFinder がある．CCFinder は研究チームの一員も携わったツールであり，国内外含めて 300 箇所以上で試用されている．コードクローン検出ツールの比較を行った研究でも CCFinder は高い再現率を示していると評価されている．しかし，CCFinder が検出するコードクローンの中にも冗長なものがある．また，処理が高速なツールであるが，一部詳細な解析を行っているため，数千万行，数億行のコードに対しては多くの時間を要する．そこで，細粒度でなるべく多くのコードクローンを高速に検出することを目指し，具体的な目標として，CCFinder よりも再現率，適合率が高いコードクローン検出手法，ツールの実現を目指した．

冗長なコードクローンとして，ソースコード中に現れる繰り返し部分に着目した．既存の行単位や字句単位の検出手法には，ソースコード中の同じ命令が繰り返し記述された部分（以降，繰り返し部分と呼ぶ）において，冗長なコードクローンを検出する，並びに検出すべきであるコードクローンを検出できない，という課題が指摘されている．例えば，

図 2 では switch 文が 2 箇所にあり，それぞれ 5 つと 3 つの case エントリを含んでおり，これら 8 つの case エントリは，リテラルが違うのみの繰り返し構造となっているとする．この場合，この switch 文に対して既存の検出ツールを適用すると 6 つものコードクローンのペアを検出してしまう．しかし，ユーザが必要とする情報は，switch 文の case エントリで実施している処理が似ている場合に，switch 文全体をコードクローンのペアとして検出することが望ましいと考えられる．本テーマでは，冗長なコードクローンを繰り返し部分を含むコードクローンと考え，より有益なコードクローン検出結果を得るために，ソースコード上の繰り返し構造を折りたたむという検出の前処理を行った上でコードクローン検出する手法を提案する．繰り返し部分を折りたたむことで，着目する必要のない冗長なコードクローンの検出を抑止するとともに，把握すべきコードクローンが見つかることが期待される．

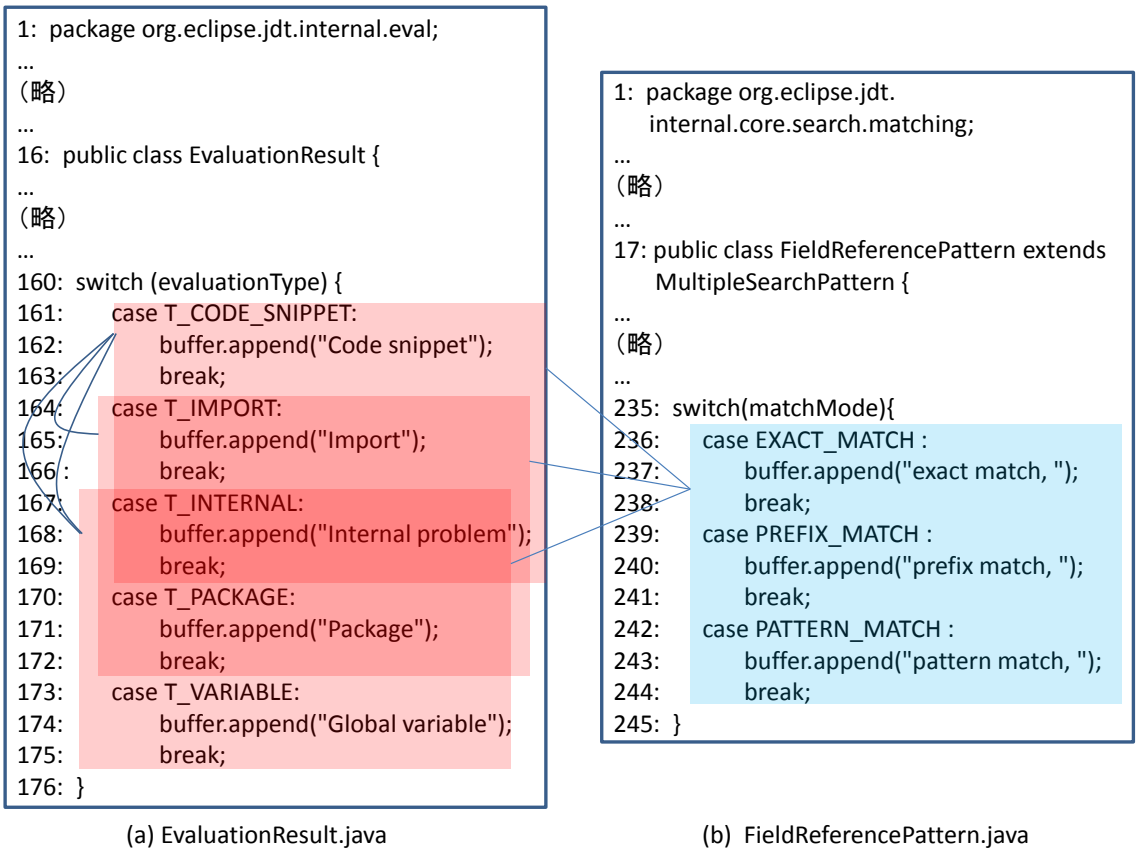


図 2　繰り返し部分に対するコードローン検出

提案手法に基づくプロトタイプシステムを開発し，幾つかのオープンオースソフトウェアと IT Spiral 実プロジェクトデータに適用して，従来手法と提案手法で検出されたコードクローンの質（再現率・適合率）を比較した．その結果，すべての適用対象ソフトウェアについて，折りたたみ処理を行うことで，検出されたコードクローン数が削減できた．すなわち，冗長なコードクローンの削減が達成できたと考えられる．一方，再現率と適合率については，適合率は全ての対象について向上したが，再現率については向上したもの

と悪くなったものが見られた（図 3，図 4 参照）．また，複数のオープンソースソフトウェアに対する検出結果の平均値で，CCFinder よりも早くコードクローン検出が行えた．

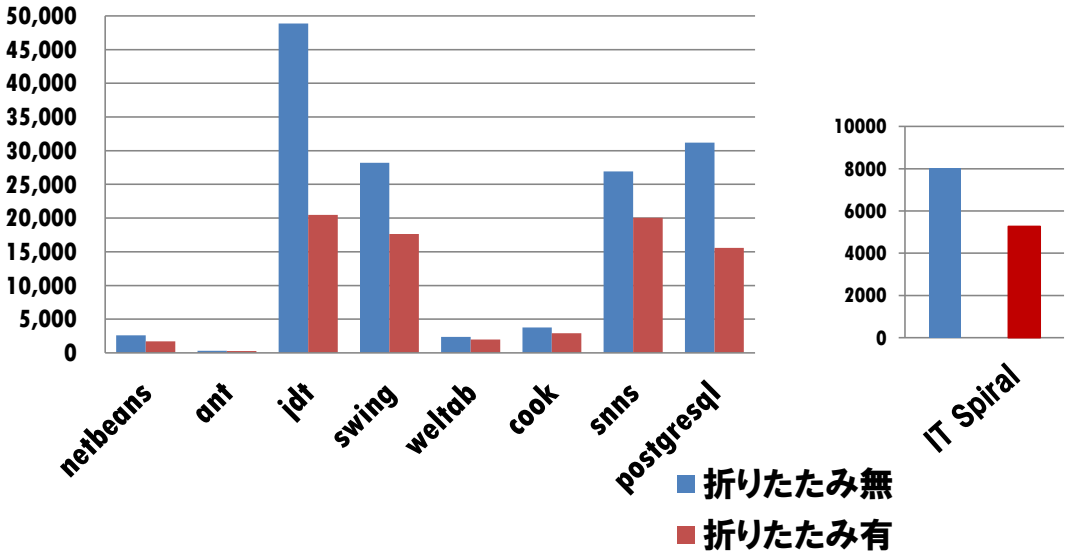


図 3　折りたたみ有・無でのコードクローン検出数

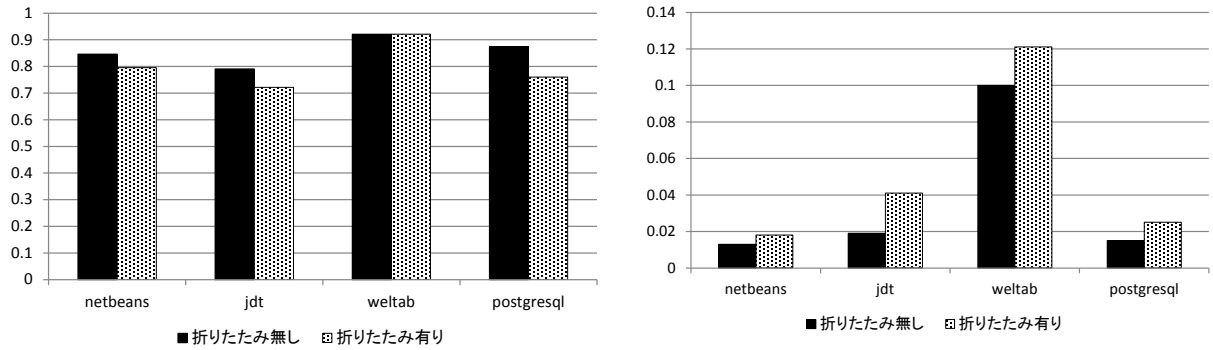


図 4　再現率と適合率

５．リファクタリング支援

次に (G2) リファクタリング支援であるが，ここでは集約しやすいコードクローン分析・対処手法を開発した．コードクローンへの対策の一つは集約である．集約とはコードクローンとなっているコード片を 1 つのメソッドなどにまとめることである．集約により保守の対象となるコードクローンの存在を除去することが可能となる．

集約の方法としては，コードクローンを検出して，いわゆるリファクタリングパターンを適用するということが考えられる．コードクローンに対するリファクタリング手法としては，Extract Method(メソッドの抽出)，Pull Up Method(メソッドの上位階層への引き上げ)等がよく行われるが，当然ながらリファクタリングパターンはコードクローンを対象として提案されているものではないので，コードクローンの特徴によっては簡単にリファク

タリングができないこともある．また，コードクローン集約作業の初級者が既存のリファクタリングパターンを参照して集約作業を行う際に，途中で集約作業を中止する，もしくはバグを含む修正を行ってしまう状況も考えられる．

一つの解決方法として，コードクローンの特徴に応じた集約方法のガイドラインの作成やリファクタリングに適したコードクローンの自動抽出が考えられる．研究チームでは過去に，特定のリファクタリングパターンに適用可能なコードクローンの分析ツールを開発してきている．この知見を生かし，より多くのリファクタリングパターンに対応できるツールの開発,集約方法のガイドラインの開発を目指した.具体的には，Template Method パターンと呼ばれるデザインパターンに着目した．Template Method パターンとは，共通の親クラスを持つ類似メソッドを対象とし，メソッド間で共通の処理を親クラスに記述し，メソッドごとに異なる処理を子クラスに記述する，というパターンである（図 5）．このパターンを用いたコードクローン集約手法では，メソッド間でコードクローンとなっている箇所を共通の処理として親クラスに引き上げ，コードクローンとなっていない箇所を子クラスに記述することで，コードクローンの集約を実現している．この手法の最大の特長として，対象となるメソッドがコードクローンとなっていない箇所を含んでいても適用が可能である，という点が挙げられる．この手法に基づく支援手法が幾つか提案されている．しかし，既存手法には以下に挙げる課題点が存在する．

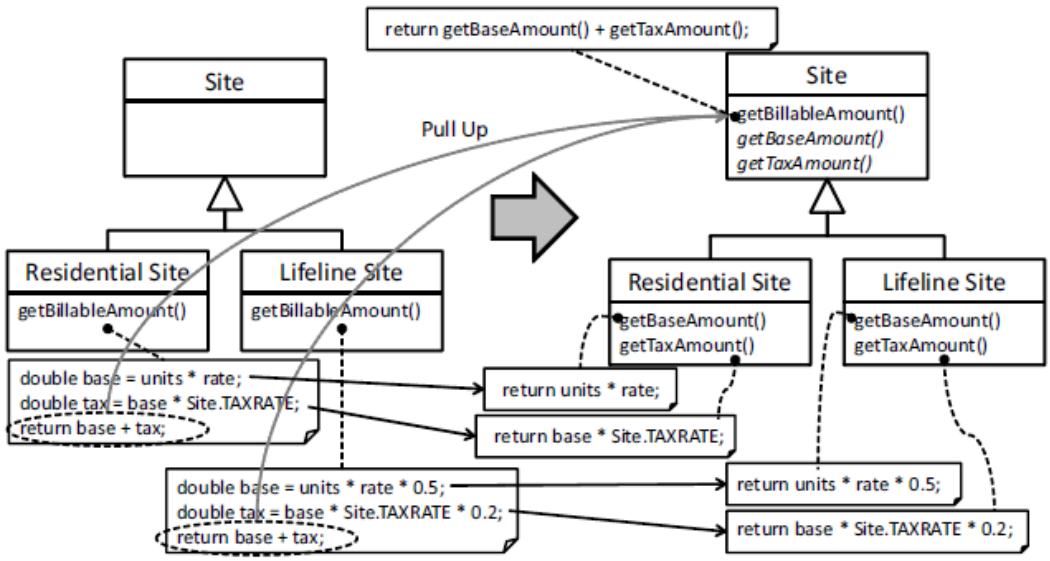


図 5　Template Method パターン

・変数名などのユーザ定義名のみが異なるコードクローンを集約できない.

・意味的に同じ処理を行っているコードでも,その表現方法が異なる場合は集約できない.

そこで本テーマではこれらの課題点を改善するため，プログラム依存グラフを用いたTemplate Method パターンの適用支援手法について実施した.具体的には,Template Method パターンの適用候補となるコードクローンを自動的に検出し，ユーザに提示するプロトタイプシステムを開発した．更に，幾つかのオープンソースソフトウェアと IT Spiral 実プ

ロジェクトデータに適用し，検出されたコードクローンの内容と検出時間を評価した．適用結果を，表 1 に示す．規模 3 万行～32 万行程度のソフトウェアに対して，表 1 に示す時間で，Template Method パターンが適用可能なコードクローンを検出できた (図 6)．また，Apache Synapse から検出した 45 個の候補に対して，Template Method パターンを適用して集約を行った．1 個あたり 10 分程度で集約を行え，集約前後で動作が変わらないことを確認した．

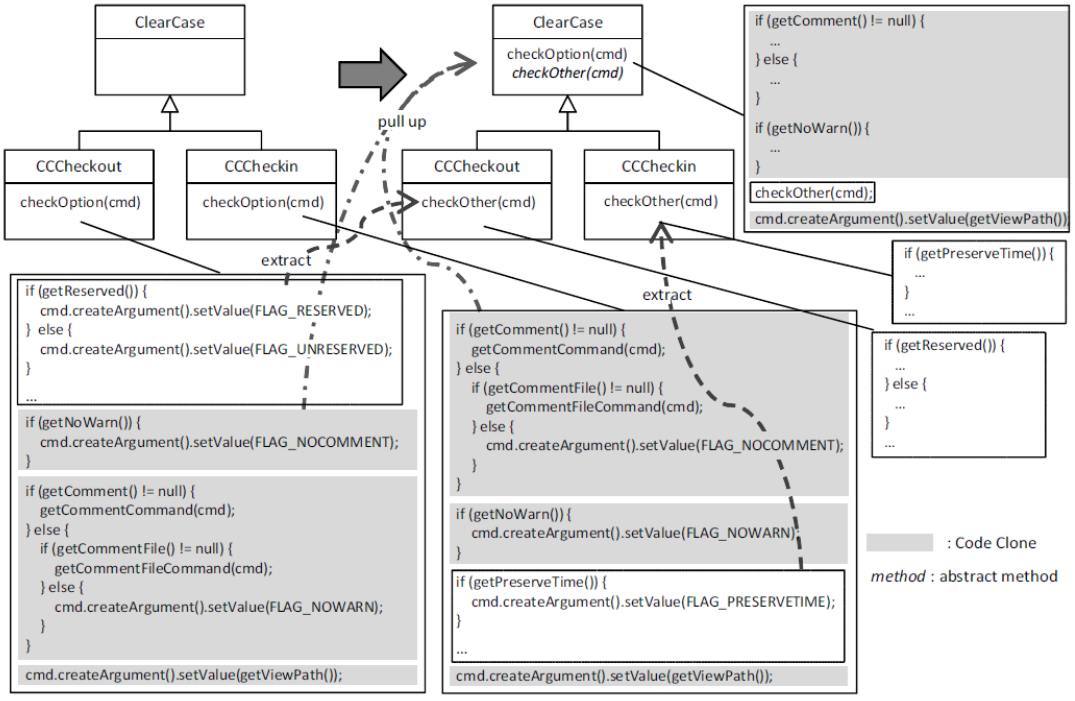


図 6　抽出例

表 1　適用結果

Target Systems	LOC	# of Candidates	Elapsed Time[s]
Apache Ant	212,401	226	237
Argo UML	328,582	486	1,080
Apache Synapse	58,418	45	95
IT Spiral	31,948	9	45

６．再利用ライブラリ支援

(G3)再利用ライブラリ作成支援では，単独で頻繁に再利用されるコード片の検出支援を行った．一般にコードクローンはソフトウェアの保守性を阻害すると言われている．一方，研究チームが過去に幾つかの企業のソフトウェアに対してコードクローン分析を行った時に，品質の高いコード片については積極的にコードクローンとして利用しているという意

見があった．しかし，品質が高いコード片であっても，将来的に変更修正が発生しないという保証は無い．従って，品質が高いコード片で，多くのプロジェクトで利用（流用，コピー）されているものは，ライブラリとしてまとめておくことが無難である．一方で，ライブラリとしてまとめる部品については，その粒度も検討する必要がある．字句/行単位で検出されるコードクローンは部品とするには小さすぎ，ファイル単位のコードクローンは粒度が大きすぎる．そこで，大規模なソースコード群から，適切な大きさ（例えば，メソッド，関数単位）のコードクローンをなるべく高速に検出するための手法とツールの開発を目指した．

７．違反流用コード発見支援

(G4)違反流用コード発見支援では,ライセンス違反等が疑われるコードの検出について検討した．近年，再利用されたソースファイルのライセンスと再利用先のソフトウェアのライセンス間で不整合が生じていることで，商用のソースコードに対しても，ソースコードの公開が求められたり,販売停止となるようなことが発生している.この問題に対して，ソフトウェア各ソフトウェアのライセンスと，再利用しているソフトウェアから到達できるライセンス集合を比較し，矛盾の有無を判定するという研究が活発に行われている．その多くは，ライセンス記述部分を用いた分析が多いが，流用されているソースコードは流用元のソースコードのコードクローンとなっていることが考えられるため，コードクローン分析は分析の一つの方法となる．しかし，ライセンス違反チェックのためには，大量のソースコードからのコードクローン検出が必要となるため，既存のアプローチではコストが非常に多くかかる．また，再利用ライブラリ作成支援でも述べた通り，適切な粒度も決める必要がある．そこで，目標としては，(G3)と同じく，大規模なソースコード群から，適切な大きさ（例えば，メソッド，関数単位）のコードクローンをなるべく高速に検出するための手法とツールの開発を目指した．結果として，(G3)，(G4）では，適用対象を Java のソースコード群，検出粒度をメソッド単位として，メソッド単位のコードクローン検出を行うプロトタイプシステムの開発を大規模 Java ソースコード群に対する適用実験を行った．メソッド単位のコードクローン検出の概略を図 7 に示す．

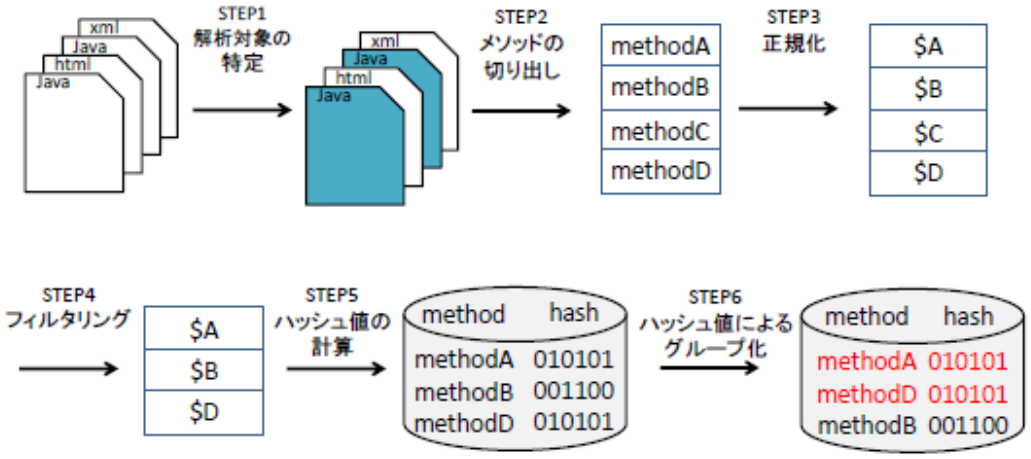


図 7　メソッド単位コードクローン検出概要

"UCI source code data sets"（ファイル数：200 万，総行数：約 4 億行）に対して提案手法を適用した結果，約 5 時間で約 81 万（要素メソッド数：約 300 万）のメソッドクローンを検出した．これらの中には，流用が疑われるメソッドクローン（図 8）や再利用ライブラリとして有用なメソッドクローン（図 9）が含まれており，本手法の有用性が確認できた．

```
39 private static Converter stringConverter =
new Converter() {
40  public Short convert(Object o) {
41   return parseShort(((String) o));
42  }
43 };

（中略）

49 public Object convertFrom(Object in) {
50  if (!CNV.containsKey(in.getClass())) throw
new ConversionException("cannot convert type: "
51    + in.getClass().getName() + " to: " +
Short.class.getName());
52  return CNV.get(in.getClass()).convert(in);
53 }

（中略）

62   CNV.put(String.class,
63    stringConverter
64   );
```

(a) mvel　ShortCHクラス

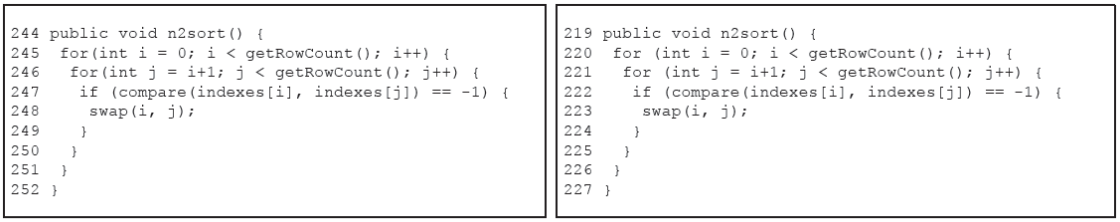
```
17 public Object convertFrom(Object in) {
18  if (!CNV.containsKey(in.getClass())) throw
new ConversionException("cannot convert type: "
19     + in.getClass().getName() + " to: " +
Short.class.getName());
20  return CNV.get(in.getClass()).convert(in);
21 }

（中略）

30   CNV.put(String.class,
31    new Converter() {
32     public Short convert(Object o) {
33      return Short.parseShort(((String) o));
34     }
35    }
36   );
```

(b) mvflex　ShortCHクラス

図 8　流用が疑われるメソッドクローン



```
244 public void n2sort() {
245  for(int i = 0; i < getRowCount(); i++) {
246   for(int j = i+1; j < getRowCount(); j++) {
247    if (compare(indexes[i], indexes[j]) == -1) {
248     swap(i, j);
249    }
250   }
251  }
252 }
```

(a) sun　TableSorterクラス

```
219 public void n2sort() {
220  for (int i = 0; i < getRowCount(); i++) {
221   for (int j = i+1; j < getRowCount(); j++) {
222    if (compare(indexes[i], indexes[j]) == -1) {
223     swap(i, j);
224    }
225   }
226  }
227 }
```

(b) Perham　TableSorterクラス

図 9　再利用ライブラリとして有用なメソッドクローン

８．まとめ

以上，ソフトウェア開発や保守の様々な活動，状況（コンテキスト）に応じた支援を目的とし，以下の 4 つのコンテキスト，(G1) ソースコード理解支援，(G2) リファクタリング支援，(G3) 再利用ライブラリ作成支援，(G4) 違反流用コード発見支援，それぞれに応じたコードクローン検出手法の開発と検出されたコードクローンに対する対策手法を提案した．また，オープンソースソフトウェアや IT Spiral 実プロジェクトデータに適用した結果，各手法の有効性が確認できた．今後の課題としては，提案手法の拡張，評価実験の継続を通じて，実際の開発現場へ普及していくことが考えられる．

１．研究の背景および目的

1.1 背景

近年，ソフトウェアシステムは社会の至る所の基盤となっており，社会生活において必須のものになっている．また，我が国における情報サービス業の売上げは平成 22 年には約 19 兆円，その多くがソフトウェア開発関連業務によるものであり[Keisan]，経済界においても重要な産業分野と認識されている（図 1-1）．また，一般家庭におけるパーソナルコンピュータの普及率も平成 21 年度には 87.2%に達しており[Soumu]，ビジネスだけでなく普段の生活においてもソフトウェアは重要な役割を果たしている．

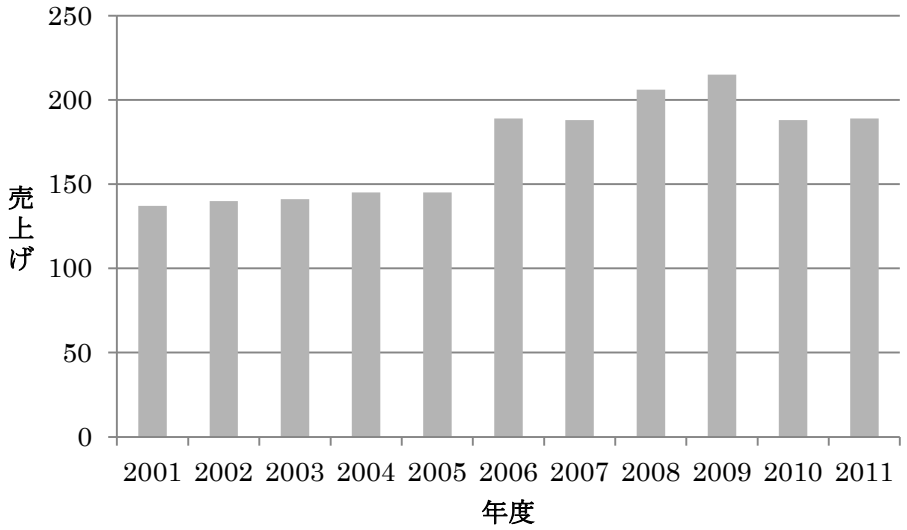


図 1-1　情報サービス業の売り上げ

一方，我が国におけるソフトウェア開発プロジェクトの成功率はあまり高くないという報告もある[Nikkei]．最近でも，特許庁の情報システム開発の失敗により約 55 億円の損失が発生しており[Tokkyo]，ソフトウェア開発に対する技術面，管理面での更なる改善が求められている．

ソフトウェア開発,保守を阻害する問題の一つとしてこの 10 年間ソースコード上のコードクローンに関する問題が提起され，活発に研究がなされている．コードクローンとは，ソースコード上に存在する同一，または，類似したコード辺を意味する．ソースコードの流用により，バグの拡散（あるコード片中にバグが存在した場合，そのコード片のすべてのコードクローンに対して，同様のバグの存在が疑われる），あるいは，機能追加時の一貫性確保の難しさ（機能追加により修正対象となるコード片のすべてのコードクローンに対して，同様の修正を実施しなければならない可能性がある）等の問題が指摘されている．特に，対象ソフトウェアが大規模な場合，チェックすべき箇所が膨大な数になってしまうこと，及び，人間がすべての重複部分を認識しておくことは難しいことから，大規模ソフトウェアからのコードクローン検出手法や分析結果の利用等が研究されてきている．

ソフトウェア工学の分野で最も歴史と権威がある国際会議 ICSE（International Conference on Software Engineering, ソフトウェア工学国際会議）[ICSE]においてもコードクローンに関するセッションが企画されている．ソフトウェア保守に関する国際会議（International Conference on Software Maintenance (ICSM)）[ICSM]においても主要な研究テーマとなっている．更に，2007 年からはコードクローンに特化した国際会議（International Workshop on Software Clones (IWSC)）も開催されている．

1.2 研究課題

ここでは，これまでに行われてきているコードクローン分析の利用方法についてのべ，本研究での課題をまとめる．

コードクローン分析の利用としては，大きく以下の４つにまとめられる．

・ソースコードの現状把握
・リファクタリング候補の抽出
・デバッグ対象コードの抽出
・潜在的不具合の検出

ソースコードの現状把握では，出荷前ソフトウェアのアーキテクチャ分析，保守対象のソフトウェア中の分布状況，規模の水増しチェック等が行われている．文献[Douyama2007]では，コードクローン検出ツール CCFinder[Kamiya2002]を用いた納品前ソフトウェアのアーキテクチャ分析が紹介されている．コードクローンが存在する場合は，その存在理由の明確化が求められている．リファクタリング候補の抽出では，集約しやすいコードクローンの抽出が行われている．コードクローンとして検出されたコード片の集合は必ずしも，関数や手続きにそのまま集約できるものとは限らない．文献[Higo2004]では幾つかのコード片の特徴を表す幾つかのメトリクスを用いて，集約しやすいコードクローンをユーザに提示する支援方法を提案している．デバッグ対象コードの抽出では，デバッグ対象となったコード片に関するコードクローンを検出することで，同じバグの取り残しを避けることが目的となる．文献[Morisaki2008]では，CCFinder がコードクローン検出エンジンとして使用されているデバッグ対象コード片検出ツール Libra を用いたデバッグ支援手法を評価している．Libra では，修正対象コード片を入力として，入力コード片に対するコードクローンを検出対象ソースコードから抽出する．結果として，修正対象コード片と同様の修正が必要なコード片の８割程度が自動的に検出されることが確認されている．コードクローンに関連する潜在的不具合の検出では，コピーされたコード片に対する修正漏れ部分で現時点では不具合として顕在化していない部分の検出を目的とする．文献[Li2006]では，コードクローン検出ツール CP-Miner を用いたコードクローン検出と検出されたコード片に対する潜在的不具合（コピーした後修正漏れ）の調査を行っている．

また，コードクローン分析技術の有望な利用先として，ライセンス違反の検出がある．元請けのソフトウェア開発組織が，開発を関連会社，あるいは，海外にアウトソーシングを行った結果，オープンソースソフトウェア等のライセンスに違反して，ソースコードを流用したものが納品された場合，元請け組織がそのリスクを負うことになる．商用のライセンス違反検出サービスも幾つか存在する[Blackduck]が,その違反検出の方法は明らかになっていない．コードクローン検出技術はソースコードの類似部分を自動検出するため，

大規模なオープンソースソフトウェア群に対してスケーラブルな手法が開発されれば，ライセンス違反検出をより明確に行うことが可能になると考えられる．

しかし，1.1 で述べた特許庁の情報システム開発失敗の原因の一つとしても，品質の悪いコードの流用によるコードクローンの作り込みが指摘されており，研究成果が開発現場にあまり浸透していないことがうかがわれる．開発現場での利用を考える場合には，一般的なコードクローン検出/分析手法の提示だけでは不十分で,利用目的や状況に特化した手法の開発とその有用性の評価結果を合わせて提示する必要があると考えられる．

本研究では，ソフトウェア開発や保守の様々な活動，状況（コンテキスト）に応じた支援を目指す．具体的には，幾つかのコンテキストに応じたコードクローン検出手法の開発と検出されたコードクローンに対する対策手法の開発を行う(図 1-2)．

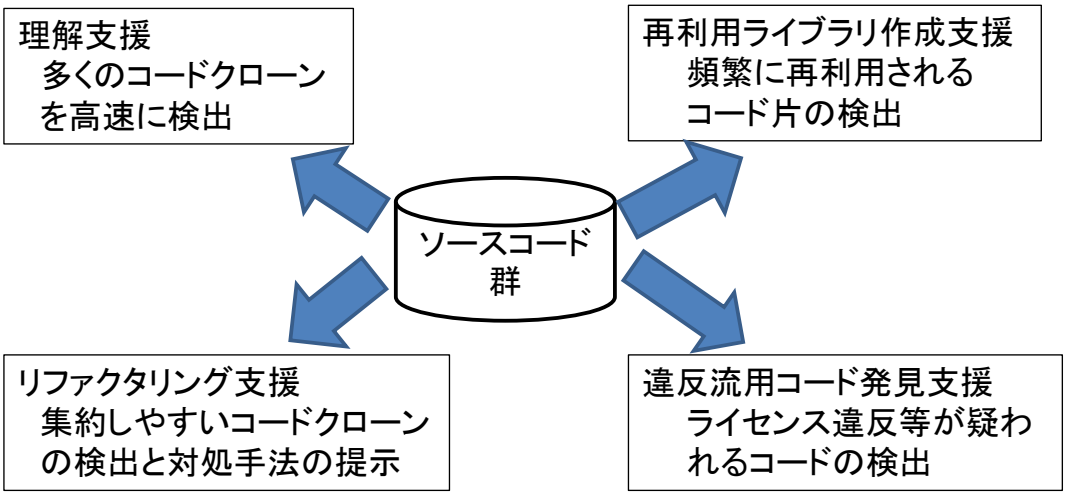


図 1-2　コードクローン分析を用いる４つのコンテキスト

1.3 研究の意義

先ず，ソースコード理解支援に関しては,近年デファクト的に使用されている CCFinder をはじめとして，行単位・字句単位のコードクローン検出手法が検出してしまう冗長なコードクローンを削減することに加えて，細粒度でなるべく多くのコードクローンを高速に検出することが可能となるコードクローン検出手法を開発することは，コードクローン分析の精度，効果を高めることになる．次に，リファクタリング支援に関しては，従来対応可能であった単純なリファクタリングパターンだけではなく，より複雑なリファクタリングパターンへの対応をすることで，リファクタリング作業の効率化に寄与することができる．再利用ライブラリ作成支援については，大規模なソースコード群から，適切な大きさのコードクローンをなるべく高速に検出するための手法とツールの開発を行うことで，企業等で蓄積された大規模ソースコード群から有用な再利用部品群を作成することができ，開発効率の改善に役立つ．最後に，違反流用コード発見支援では，ライセンス違反等が疑われるコードの自動検出により，違反流用に対するリスク削減が可能となる．

２．実施内容

2.1 研究アプローチ

2.1.1 研究の全体像

1.2 の図 1-2 に示す４つのコンテキストを想定し研究を行う．以降，それぞれについて述べる．

(1) ソースコード理解支援：細粒度でなるべく多くのコードクローンを高速に検出する手法

これまでに多くのコードクローン検出手法が提案されている．その手法は，ソースコードからコードクローンを，行/字句単位で一致行/字句列として検出，グラフ(プログラム依存グラフ，あるいは，抽象構文木)に変換して一致部分グラフとして検出，メトリクスを用いて特徴が一致するモジュール（関数，クラス等）単位の検出，等の手法に分けられる．それぞれの手法で多くの検出手法,ツールが開発されてきている.ツールの評価としては，再現率（正解コードクローンの中の何割をツールが検出できるか）と適合率（ツールが検出したコードクローンの何割が正解であるか）で評価されることが多い（再現率と適合率はトレードオフの関係にあるため,それらの調和平均である F 値が用いられることもある)．なるべく多くのコードクローンを検出するという立場であれば，再現率が重視される．細粒度でなるべく多くのコードクローンを検出できるツールとして著名なものに CCFinder がある[Kamiya2002]．CCFinder は研究チームの一員も携わったツールであり，国内外含めて 300 箇所以上で試用されている．コードクローン検出ツールの比較を行った研究[X5]でも CCFinder は高い再現率を示していると評価されている．しかし，CCFinder が検出するコードクローンの中にも冗長なものがある．また，処理が高速なツールであるが，一部詳細な解析を行っているため，数千万行，数億行のコードに対しては多くの時間を要する．そこで，細粒度でなるべく多くのコードクローンを高速に検出することを目指し，具体的な目標として，CCFinder よりも再現率，適合率が高いコードクローン検出手法，ツールの実現を行う．

(2) リファクタリング支援：集約しやすいコードクローン分析・対処手法

コードクローンへの対策の一つは集約である．集約とはコードクローンとなっているコード片を１つのメソッドなどにまとめることである．集約により保守の対象となるコードクローンの存在を除去することが可能となる．

集約の方法としては，コードクローンを検出して，いわゆるリファクタリングパターン[Fowler1999]を適用するということが考えられる．コードクローンに対するリファクタリング手法としては，Extract Method(メソッドの抽出)，Pull Up Method(メソッドの上位階層への引き上げ)等がよく行われるが,当然ながらリファクタリングパターンはコードクローンを対象として提案されているものではないので，コードクローンの特徴によっては簡単にリファクタリングができないこともある．また，コードクローン集約作業の初級者が

既存のリファクタリングパターンを参照して集約作業を行う際に，途中で集約作業を中止する，もしくはバグを含む修正を行ってしまう状況も考えられる．

一つの解決方法として，コードクローンの特徴に応じた集約方法のガイドラインの作成やリファクタリングに適したコードクローンの自動抽出が考えられる．研究チームでは過去に，特定のリファクタリングパターンに適用可能なコードクローンの分析ツールを開発してきている[Higo2005]．この知見を生かし，より多くのリファクタリングパターンに対応できるツールの開発，集約方法のガイドラインの開発を行う．

(3) 再利用ライブラリ作成支援：単独で頻繁に再利用されるコード片の検出支援

一般にコードクローンはソフトウェアの保守性を阻害すると言われている．しかし，研究チームが過去に幾つかの企業のソフトウェアに対してコードクローン分析を行った時に，品質の高いコード片については積極的にコードクローンとして利用しているという意見があった[Higo2007]．しかし，品質が高いコード片であっても，将来的に変更修正が発生しないという保証は無い．従って，品質が高いコード片で，多くのプロジェクトで利用（流用，コピー）されているものは，ライブラリとしてまとめておくことが無難である．一方で，ライブラリとしてまとめる部品については，その粒度も検討する必要がある．字句/行単位で検出されるコードクローンは部品とするには小さすぎ，ファイル単位のコードクローンは粒度が大きすぎる．

そこで，大規模なソースコード群から，適切な大きさ（例えば，メソッド，関数単位）のコードクローンをなるべく高速に検出するための手法とツールの開発を行う．

(4) 違反流用コード発見支援：ライセンス違反等が疑われるコードの検出

近年，再利用されたソースファイルのライセンスと再利用先のソフトウェアのライセンス間で不整合が生じていることで，商用のソースコードに対しても，ソースコードの公開が求められたり，販売停止となるようなことが発生している．この問題に対して，ソフトウェア各ソフトウェアのライセンスと，再利用しているソフトウェアから到達できるライセンス集合を比較し，矛盾の有無を判定するという研究が活発に行われている[German2010]．その多くは，ライセンス記述部分を用いた分析が多いが，流用されているソースコードは流用元のソースコードのコードクローンとなっていることが考えられるため，コードクローン分析は分析の一つの方法となる．しかし，ライセンス違反チェックのためには，大量のソースコードからのコードクローン検出が必要となるため，既存のアプローチではコストが非常に多くかかる.また,再利用ライブラリ作成支援でも述べた通り，適切な粒度も決める必要がある．

そこで，目標としては，(3)と同じになるが，大規模なソースコード群から，適切な大きさ（例えば，メソッド，関数単位）のコードクローンをなるべく高速に検出するための手法とツールの開発を行う．

2.1.2 関連するこれまでの研究について

(1) 本研究以前に実施されていた委託研究に関連する研究について

4つのテーマそれぞれについてまとめる.

① ソースコード理解支援

ソースコードの理解は，ソフトウェア保守を行う上で基本となる作業である．機能拡張をする場合,拡張対象の機能がソースコードのどこで実現されているのか,あるいは,特定した部分を修正した場合，全体として正しく動作するかどうかを確認する必要がある．また，バグを修正する場合も，その原因がどこにあるか，何故そのようなバグが入ったかということを確認しなければならない．これらのソースコード理解を支援する方法としては，プログラム解析に基づく支援がよく行われている．

一つのアプローチとして影響波及解析を行う手法がある[Ren2004] [Ryder2001]．影響波及解析とは，ソフトウェアのある部分が変更される際，その変更によって異なる動作をする可能性のある部分（被影響部分）を，ソフトウェア全体から特定するという解析である．ある修正を行った際，被影響部分を特定することで，被影響部分から当該修正の内容に合わせて別の修正を行う必要があることを容易に発見可能となり，また，修正後のテスト実行の際，修正箇所と被影響部分だけの動作を確認すればよい（いわゆる回帰テスト）というメリットがある．

コードクローンも一つの影響波及解析であると言える．直接的な依存関係はない場合も多いが，一つのコード片を変更するとそのコード片のコードクローンについても変更の必要性を確認する必要があるからである．保守対象のソースコード上にどの程度のコードクローンが存在するかを評価することは，ソースコード理解を行う上で重要な情報となりうる．コードクローン検出手法については3.1.2でまとめる．

② リファクタリング支援

既に述べた通り,リファクタリングとはソフトウェアの外部的振る舞いを保ったまま,ソフトウェアの内部構造を改善することによりソフトウェアの設計品質を高める技術である．しかし，どのようなときにリファクタリングが要求されるかについての厳密な基準は存在しない.Fowler は過去の経験や知識をもとにリファクタリングの可能性を示すいくつかの兆候(不吉な匂い) を定義している．また，それらの不吉な匂いのいくつかはモジュールの行数やサイクロマチック数などの複雑度メトリックスを用いて定量的に示すことができる．例えば，文献[Kataoka2002]では，モジュール間の結合度合が強い部分に対して，リファクタリングを行うことで，ソフトウェアの保守性に与える効果を計測している．また，文献[Hatano2003]では複数のソフトウェアメトリクスを用いて，ソースコードの構造的欠陥を検出し，その部分に対して適切なリファクタリングを行う手法を提案している．

コードクローンは，不吉な匂いの中の重複したコードに対応するため，重複部分を一つに集約するという支援が適切であり，検出したコードクローンとなっているメソッドの抽出や親クラスへの引き上げといったシンプルなリファクタリングが適用されている．

③ 再利用ライブラリ作成支援

文献[McMillan2011]では，コードサーチエンジン“Portfolio”を利用した有用コードの検索方法について述べられている．Portfolio ではリポジトリ中のソースコードの関数呼び出し関係をグラフで表現し，与えられたキーワードととの関連や関数間の呼び出し関係を解析することで，関数の有用度順に提示するツールである．直観的にはよく使われており，検索キーワードにマッチするものを有用コードとして抽出する．文献[Zhao2004]では，文書とソースコードの関連を求め，機能を実装しているモジュール群を文書から特定し，特定したモジュール群を元にコールグラフを用いて機能の抽出を行うという手法を提案している．

コードクローンとなっているコードは，多くの場所で利用されていることになるので，有用な部品の候補となる可能性は高い．コードクローンに着目した有用部品の抽出に関しては， 3.3 にてふれる．

④ 違反流用コード発見支援

違反有用コード発見支援手法としては，オープンソースソフトウェアのライセンス違反による再利用の検出についての研究がある[German2010]．また，商用のサービスとして，Black Duck[Blackduck]，Palamidia[Palamidia] がある．また，文献[Prechelt2002]では，コードクローンを利用した剽窃発見手法について述べている．

(2) 研究責任者が本委託研究以前に実施していた委託研究に関連する研究について

ここでは，研究グループがこれまでに実施してきたコードクローン分析に関する研究成果についてまとめる．研究内容は，コードクローン検出技術，コードクローン可視化技術，コードクローン集約技術，デバッグ支援技術にまとめられる．

① コードクローン検出技術

字句単位でコードクローンを検出するコードクローン検出ツール CCFinder を開発してきている．CCFinder の特徴を以下に述べる．

・入力対象プログラム： C， C++， Java，Cobol プログラム．

・出力：対象プログラム中に含まれる同形の部分を検出し，出力する．同形の部分とは，以下で述べる字句解析，変形ルール適用，パラメータ変換を経て同形判定アルゴリズムで同じと判定される字句系列に対応した，対象部分を言う．

・検出方法：以下の Step1～Step5 を順に行う．

Step1: ソースコードを入力し，トークン（字句）の列にする．

対象プログラム中から，字句を抽出する．この段階で，空白文字，改行文字，コメント文字，制御文字など字句の構成に関係のない文字は消去する．例えば，図 2-1 の C++プログラムに対し，図 2-2 が得られる．

```
1   void print lines(const set<string>& s) {
2      int c = 0;
3      set<string>::const_iterator i
4         = s.begin();
5      for (; i != s.end(); ++i) {
6         cout << c << ", "
7            << *i << endl;
8         ++c;
9      }
10  }
11  void print table(const map<string, string>& m) {
12     int c = 0;
13     map<string, string>::const iterator i
14        = m.begin();
15     for (; i != m.end(); ++i) {
16        cout << c << ", "
17           << i->first << " "
18           << i->second << endl;
19        ++c;
20     }
21  }
```

図 2-1　入力例

```
1   void print lines ( const set < string >& s ) {
2   int c = 0 ;
3   set < string > :: const iterator i
4   = s . begin ( ) ;
5   for ( ; i != s . end ( ); ++ i ) {
6   cout << c << " , "
7   << * i << endl ;
8   ++ c ;
9   }
10  }
11  void print table ( const map < string , string >& m ) {
12  int c = 0;
13  map < string , string > :: const iterator i
14  = m . begin ( ) ;
15  for ( ; i != m . end ( ) ; ++ i ) {
16  cout << c << ", "
17  << i -> first << " "
18  << i -> second << endl;
19  ++c;
20  }
21  }
```

図 2-2　変換例(1)

Step2:変形ルールにより，トークン列を変形する．

対象プログラムに応じて，字句の削除，変形を行う．例えば，クラス名やパッケージ名が付加的につけられている場合にそれらを省略した形と同形になるよう，クラス名やパッケージ名に相当する字句を削除する．また，プファイルや関数，手続きの切れ目など，論理的・物理的な切れ目を超えて同形部が存在しないよう，切れ目の終わりに排他的な名前の字句を付加する．図 2-3 に図 2-2 の適用結果を示す．

```
1   void print lines ( const set & s ) {
2   int c = 0 ;
3   const iterator i
4   = s . begin ( ) ;
5   for ( ; i != s . end ( ) ; ++ i ) {
6   cout << c << ", "
7   << * i << endl ;
8   ++ c ;
9   }
10  }
11  void print table ( const map & m ) {
12  int c = 0 ;
13  const iterator i
14  = m . begin ( ) ;
15  for ( ; i != m . end ( ) ; ++ i ) {
16  cout << c << ", "
17  << i -> first << " "
18  << i -> second << endl ;
19  ++ c ;
20  }
21  }
```

図 2-3　変換例(2)

Step3:パラメータ置換を行う.

ユーザ定義名を同一の字句として認識するよう特定の字句名に変換する．変換例を図 2-4 示す．ユーザ定義名は＄p に置き換えられている．

```
1   $p $p ( $p $p & $p ) {
2   $p $p = $p ;
3   $p $p
4   = $p . $p ( ) ;
5   for ( ; $p != $p . $p ( ) ; ++ $p ) {
6   $p << $p << $p
7   << * $p << $p ;
8   ++ $p ;
9   }
10  }
11  $p $p ($p $p & $p ) {
12  $p $p = $p ;
13  $p $p
14  = $p . $p ( ) ;
15  for ( ; $p != $p . $p ( ) ; ++ $p ) {
16  $p << $p << $p
17  << $p -> $p << $p
18  << $p -> $p << $p ;
19  ++ $p ;
20  }
21  }
```

図 2-4　変換例(3)

Step4:マッチングアルゴリズムによりクローンを検出する.

出力の字句系列のなかから同じ部分系列を持つものを発見する．図 2-5 は，2-4 のプログラムの一部で，点のある部分が同じ字句であることを示している．一定数以上，同じ字句の連続する部分は，右下下がりの直線として表現される．

Step5:クローンの位置(ファイル，行)を出力する.

発見されたクローンが，元のプログラム上のどこに相当するかを辿り，元のプログラム上の位置を，出力する．この例の場合，1-7 行と 11－17 行がクローン，また 8－10 行と 19－21 行がクローンと判定される．

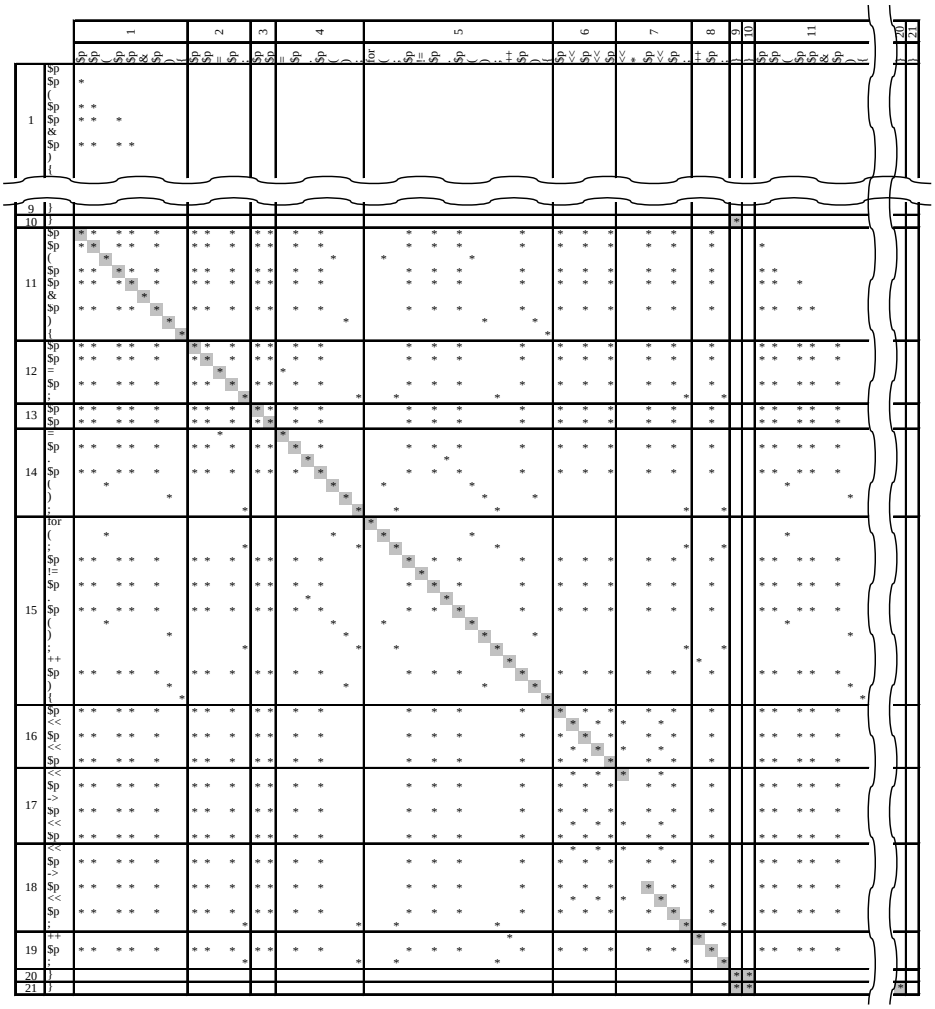


図 2-5　クローン検出例

・CCFinder の評価

数万～数百万ステップのオープンソースソフトウェアへの適用，実際のプロジェクトでの適用結果，教育機関でのソフトウェア開発で適用を行ったきた．その他，国内外 300 箇所以上の組織で試用されている．以下，幾つかの代表的な適用例について紹介する．

・JDK のライブラリへの適用

JDK（Java Development Kit） 1.2.2（サンプルとデモプログラムを除く）に対しての適用結果を図 2-6 に示す．入力ファイルは 1648 個，約 50 万行で，実行には，約 3 分を要した．図 2-6 は，両軸はソースファイルを辞書順に並べたものであり，20 行以上のクローンを図示している．(A)の部分に多くのクローンが密集しており，(B)の部分に最長のクローンが検出された．

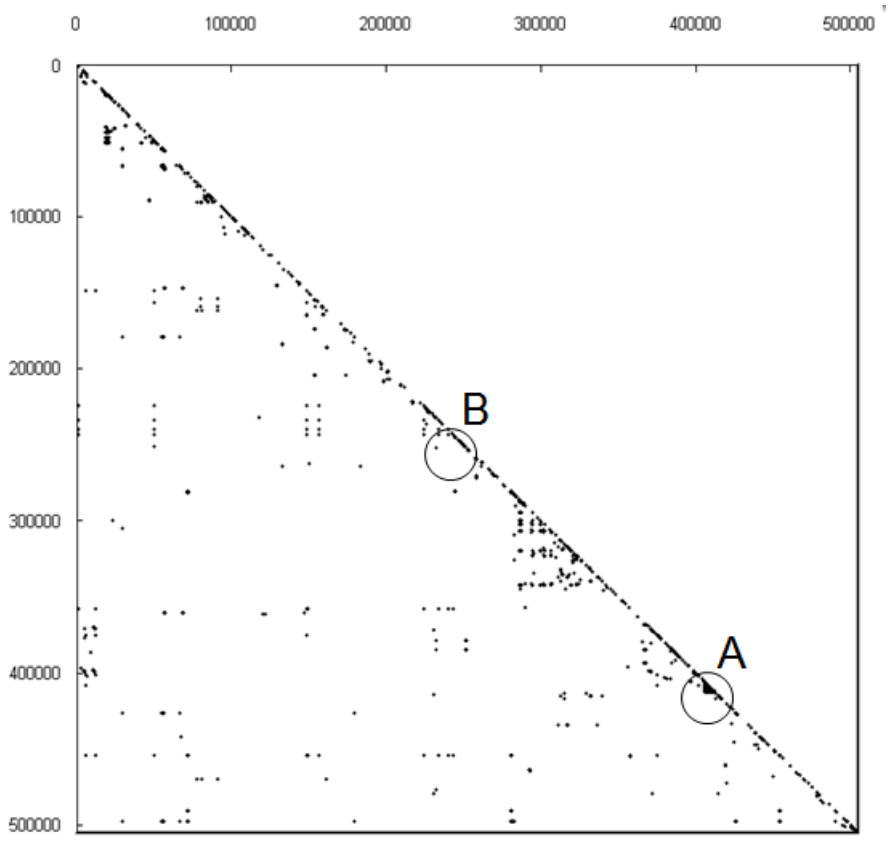


図 2-6　JDK1.2.2 への適用結果

(A)の部分は src/javax/swing/plaf/multi/*.java の 29 のファイルであり，コード生成ツールによって生成されたものであった．いくつかはクラス名を除いてまったく同じクラスの定義を含んでいた．(B)の部分は，最長のクローン（349 行）であり，src/java/util/Arrays.java の 18 の“sort”メソッドであった．シグネチャ（引数の型と数）が異なるが，アルゴリズムは同一のものであった．

・FreeBSD，Linux，NetBSD への適用

FreeBSD 4.0 (C 220 万行)，Linux 2.4.0 (C 240 万行)，NetBSD 1.5 (C 260 万行)の 3 つの UNIX 系オペレーティングシステム間のコードクローン検出を行った．結果を，図 2-7 に示す．検出には約 2 時間を要した．

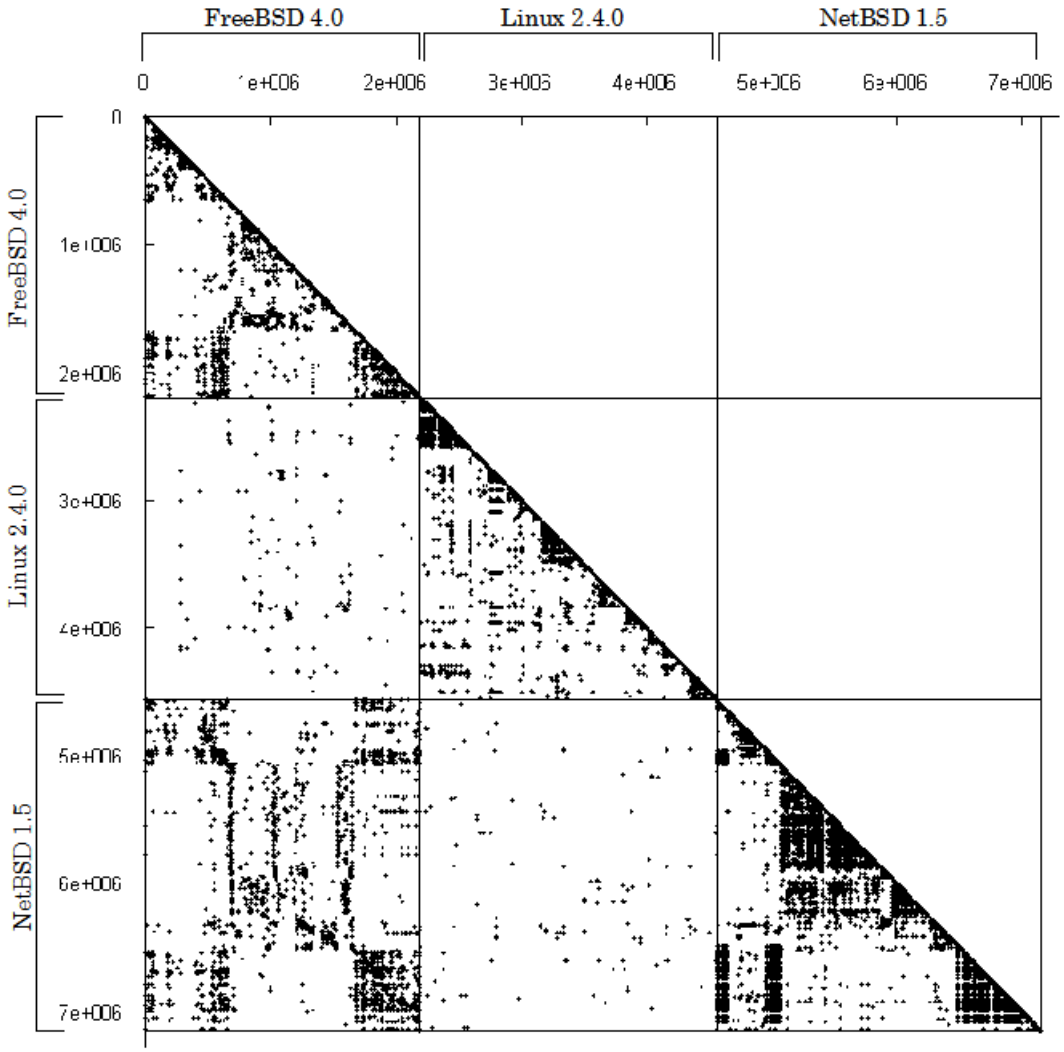


図 2-7　FreeBSD, Linux, NetBSD のコードクローン散布図

図 2-7 に示すとおり，FreeBSD と NetBSD 間には，多くのクローンが見られる．一方，FreeBSD と Linux，NetBSD と Linux 間にはあまり多くのクローンは見られない．FreeBSD と Linux 間で見られるクローンの主なものは，デバイスドライバ関係のコードであった．例えば，共通のソースから分岐したソースファイル，名前が付け替えられたソースファイル，あるソースファイルを複数のファイルに分割しているもの等が確認できた．

・実際のプロジェクトデータへの適用

文献[Higo2007]において，情報処理推進機構（IPA）技術本部ソフトウェア・エンジニアリング・センター（SEC）の先進ソフトウェア開発プロジェクトにおいてベンダ 5 社が共同でプローブ情報システムに対して適用した[Matsuura2006]．各社は個別に開発を行っており，プロジェクトマネージャでもソースコード，開発工数，開発体制（外部委託先，要員数等）に関しては知りえない状況であった．プロジェクトマネージャが主催する進捗会議では，各社のマネージャから各工程の進捗割合と，予定日数のず

れが報告されるのみであった．このようなブラインドマネジメントを支援するために，つまりブラックボックスとなっているソースコードの特徴を把握するために，コードクローン分析を行った（なお，この分析は共同開発各社合意の下に確保されたソフトウェア工学研究用の機密室内で行った）．適用は単体テスト終了後，結合テスト終了後の 2 回行った．全社合計の開発規模は数十万ステップであり，結合テスト後は単体テスト後に比べ約 2 万ステップ増加していた．このシステムは C/C++言語を用いて開発されている．コードクローン分析は各社が開発したソースコードに対して個別に行った．この実験では，30 字句以上のコードクローンを検出した．なお，検出された全てのコードクローンは関数内に閉じたものである．検出例を図 2-8 に示す．

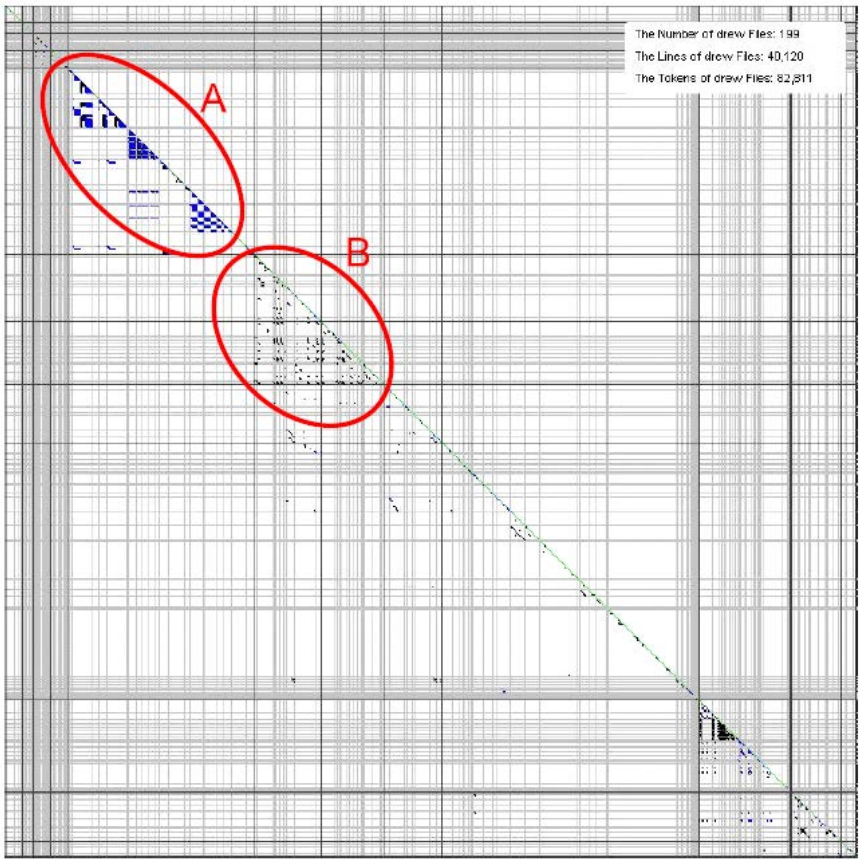


図 2-8　検出例

図 2-8 中の．A の部分にコードクローンが多数存在していた．これらのソースコードを閲覧したところ，デバック用の情報出力コード（連続した printf 文）やデータの妥当性チェック（連続した if 文）であった．B の部分にも多くのコードクローンが存在した．この部分のコードクローンは黒の格子をまたいで存在することからディレクトリ間クローンであることがわかる．この部分は車両の位置情報を扱う処理部分であり，車両種別によりディレクトリが分かれていた．ディレクトリ毎に扱う情報の種類は異なるものの，処理内容は非常に類似していた．

② コードクローン可視化技術

CCFinder の試用が増えるにつれ，多くのフィードバックが得られ，幾つかの改善点が明らかになった． CCFinder はコマンドラインツールであり，コマンドプロンプトで，引数として対象ソースコードや検出オプションを指定することで，出力ファイルにコードクローン情報を出力していた．１つめの問題はその CCFinder の引数の複雑さであった．対象ソースコード群，出力ファイル，最小一致トークン数，メモリ量などすべての情報を引数として与えなければならないため，そのコマンドは必然的に複雑になっていた．２つめの問題は CCFinder の検出したコードクローン情報であった． CCFinder は図 2-9 のようなコードクローン情報を出力する． CCFinder はこのように対象ファイルのどの部分がクローンになっているかを，行数・列数で表現している．ユーザがこの情報から，ソースコードを閲覧するためにエディタプログラムなどを起動し，ファイルのその部分まで移動する，というのは非常に手間を要した．

```
#version: ccfinder 7.2.4
#format: pairwise
#langspec: C
#option: -b 30
#option: -e char
#option: -k 30
#option: -r abdfikmnpstuv
#option: -c wfg
#begin{file description}
0.0         249         697         C:¥experiment¥linux-2.6.5¥fs¥autofs¥dirhash.c
0.1         52          93          C:¥experiment¥linux-2.6.5¥fs¥autofs¥init.c
0.2         249         622         C:¥experiment¥linux-2.6.5¥fs¥autofs¥inode.c
0.3         557         1591        C:¥experiment¥linux-2.6.5¥fs¥autofs¥root.c
0.4         30          58          C:¥experiment¥linux-2.6.5¥fs¥autofs¥symlink.c
0.5         205         468         C:¥experiment¥linux-2.6.5¥fs¥autofs¥waitq.c
1.0         272         656         C:¥experiment¥linux-2.6.5¥fs¥autofs4¥expire.c
1.1         42          63          C:¥experiment¥linux-2.6.5¥fs¥autofs4¥init.c
1.2         315         774         C:¥experiment¥linux-2.6.5¥fs¥autofs4¥inode
                        :
                        :
#end{file description}
#begin{clone}
0.2     73,2,116    86,16,165   47      1.2     124,2,241   137,18,290  47
0.2     152,2,385   165,6,420   34      1.2     221,2,547   237,6,582   34
0.2     171,2,432   201,2,521   81      1.2     248,2,598   282,2,687   81
0.2     80,3,140    88,5,173    32      2.13    193,3,397   201,6,430   32
0.2     75,2,118    88,5,173    52      4.6     702,2,1555  715,6,1610  52
0.2     75,2,118    85,15,161   41      13.10   317,2,659   327,19,702  41
0.2     75,2,118    85,15,161   41      14.11   609,2,1273  619,19,1316 41
0.2     75,2,118    83,47,153   33      15.4    353,2,780   361,44,815  33
                        :
                        :
#end{clone}
```

図 2-9 CCFinder 出力例

そこで，ユーザインターフェースの問題を解決するために， Gemini が開発された [Ueda2002a] [Ueda2002b] [Ueda2003]． Gemini は CCFinder の GUI フロントエンドプログラムであり，ユーザは Gemini 上で検出オプションを設定し CCFinder を実行する． Gemini により，CCFinder をコマンドライン上で実行する必要がなくなった．また，Gemini は CCFinder の出力ファイルを読み込み，コードクローン情報を視覚的に表示する（図 2-10）． Gemini を用いることでユーザは対象ソフトウェアのどの部分がクローンなっているか，また，他のどの部分とクローンになっているかということを瞬時に把握することが可能となった． Gemini にはソースコードを表示する機能もあり，コードクローンのソースコードを閲覧するために，エディタなどを起動す

る必要もなくなった． また，Gemini の機能として差分を含むコードクローン検出に関する研究も実施した[Ueda2002c]．

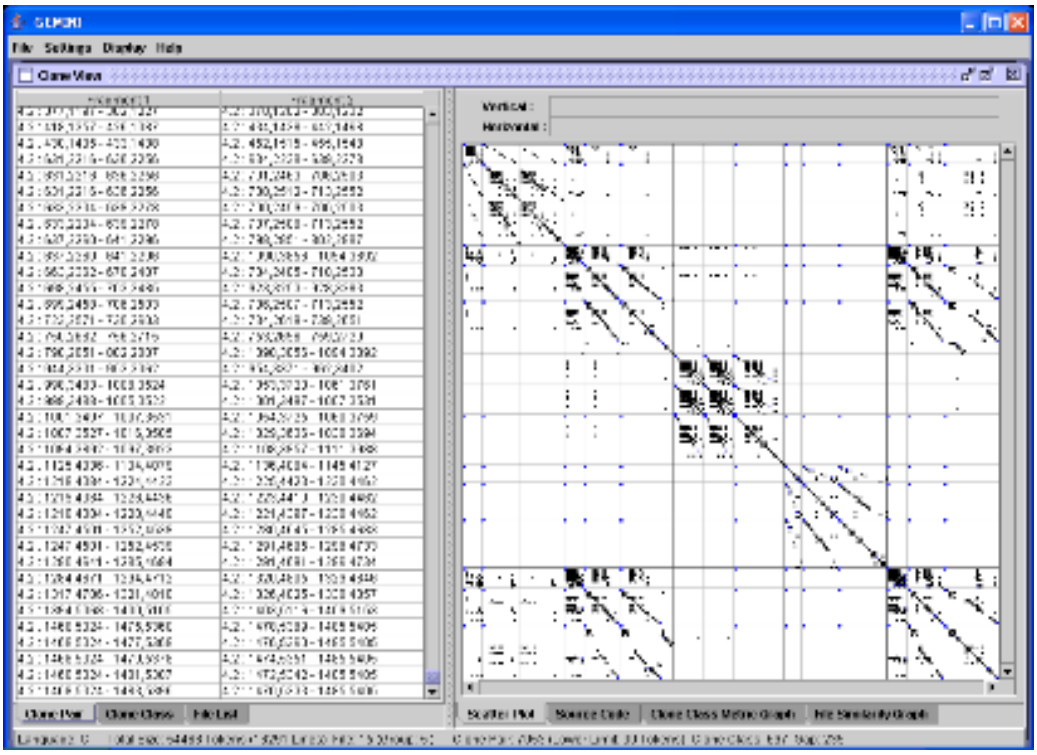


図 2-10 Gemini 出力画面

③ コードクローン集約技術

CCFinder の検出したコードクローンは，連続したトークンであるため，そのままでは集約を行う単位としては不適切であった．そこで，CCFinder の検出したコードクローンから，構造的なまとまりのある部分（クラスやメソッドなど）を抽出し，集約の支援をするためにコードクローン集約支援ツール Aries の開発を行った[Higo2005]．

Aries では，先ず，CCFinder で検出されたコードクローンから，構造的なまとまりの部分を抽出する．次に，それらがどの様に集約可能であるかを予測するために，各抽出部分をメトリクスを用いて，定量的に特徴付ける．Aries で用いている代表的なメトリクスはコードクローン間の結合度と分散度である．直観的には，結合度が高い場合，コードクローンとなっているコード片がその外部で定義された変数を使用していることになり，コード片をソフトウェアの他の部分に移動することが困難であることが予想される．一方，分散度が高い場合は，コード片が複数のクラスに存在することになり共通化が難しいと判断される．

Aries は上記の処理を行った後，解析結果をファイルに出力する．ユーザはそのファイルを Aries の GUI 部に読み込ませることにより，グラフィカルにリファクタリング支援を行うことができる．図 2-11 は Aries の画面例である．ユーザは，ウィンドウの左上のタブで，単位を宣言，メソッド，文の中から選択する． Aries ではウィンドウの左側にメ

トリクスグラフとクローン単位選択パネルが存在する．メトリクスグラフは，各メトリクス値に基づいたクローンの絞込みを行うことができる．また，クローン単位選択パネルでは，クローンの単位に基づいた絞込みを行うことができる．図 2-11 は文単位のタブの様子を表す．文単位のクローンは，do，for，if，switch，synchronized，try，while の 7 つがある．クローン単位選択パネルでは，それぞれの単位についてチェックボックスがあり，絞込みの対象とするかを選択できる．メトリクスグラフとクローン単位選択パネルの絞込み結果は，ウィンドウ右側のクローンセットリストに反映され，絞込みの結果，該当するクローンセットのみがクローンセットリストに表示される．

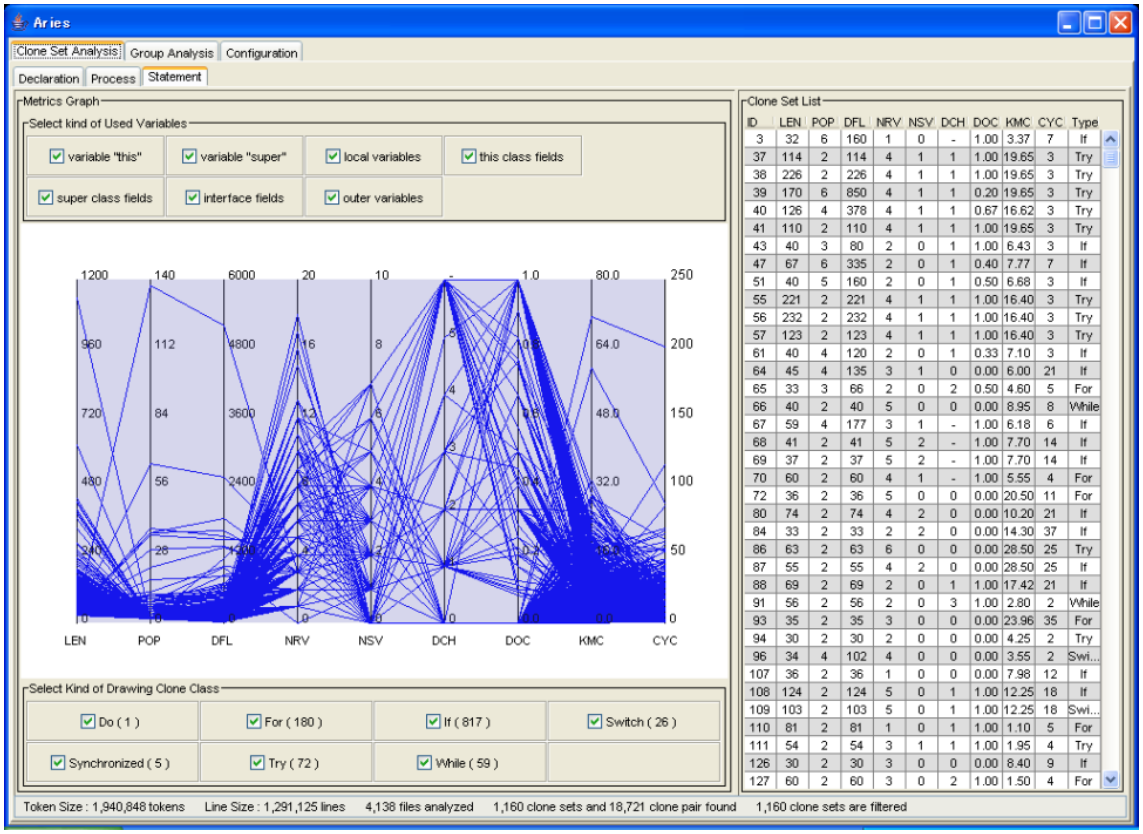


図 2-11　Aries 出力例

④　デバッグ支援技術

コードクローンに対するデバッグ時や機能追加時の修正支援を目的として，Libra を開発した[Izumida2003]．Libra では内部で CCFinder のオプションを用いて，ユーザが入力したコード片と対象ソフトウェアの間のコードクローンのみをユーザに提供する．CCFinder は以下の 3 タイプのコードクローンの検出のオン･オフがオプションで設定できる．

ファイル内クローン

ファイル間クローン

グループ間クローン

Libra はユーザが入力したコード片をグループ1,対象ソースコードをグループ2として，グループ間クローンのみを検出するオプションをつけて，CCFinder を実行する．このように CCFinder を実行することによって，必要最小限のコードクローンのみを検出することができる． CCFinder は全てのコードクローンを検出した後，それらをソーティングして出力を行う．必要最低限のコードクローンのみを検出することで，ソーティングにかかる時間を短縮することができ，より高速に検出結果を得ることができる．

図 2-12 が Libra の起動画面である.Libra はここで入力されたコード片と指定されたソースコード間のコードクローンのみを検出する．ユーザはこの部分で，最小一致トークン数を入力したコード片のトークン数にすることができる．このようにすることで，入力されたコード片の一部分のみが類似しているコードクローンの検出を制限することができる.

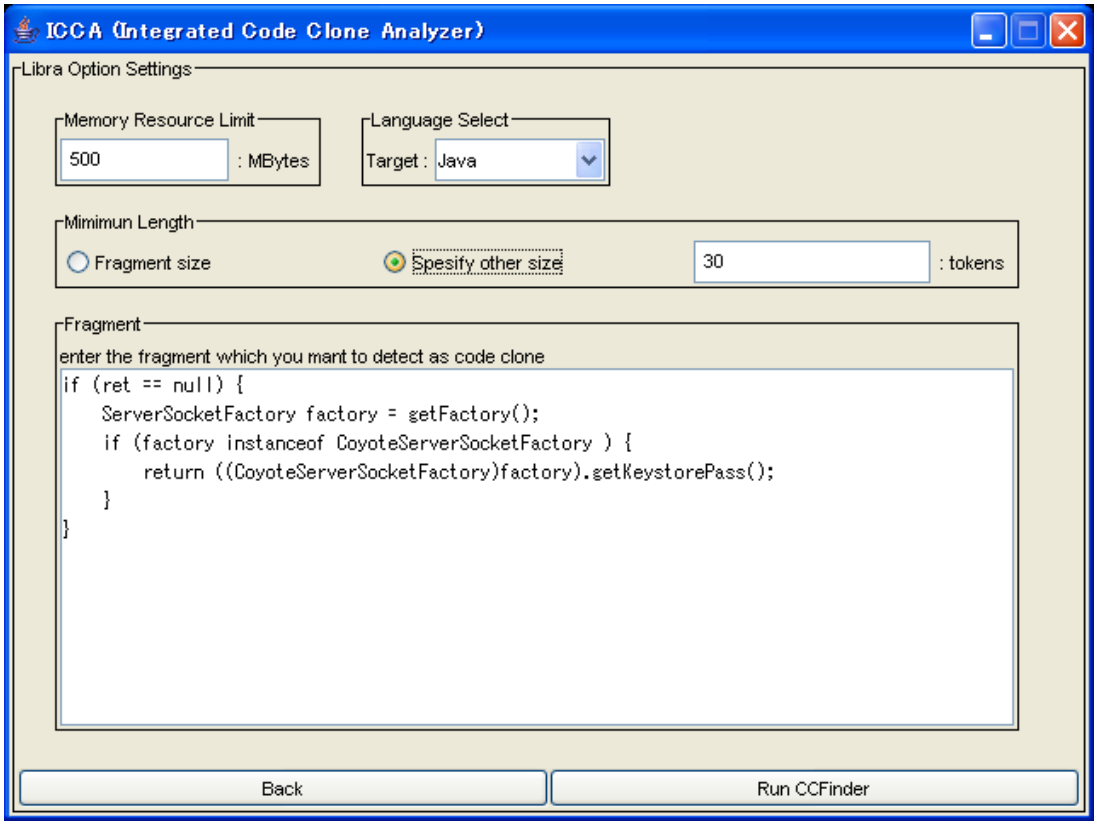


図 2-12　Libra 起動画面

ユーザが，コード片とソースコードを指定すると Libra が CCFinder を実行してコードクローン検出を行う．解析が終わると，Libra の GUI が立ち上がる（図 2-13）．図 2-13 では，ウィンドウの左側に対象ソースコードのディレクトリツリーが表示される．指定されたコード片とクローンになっているソースコードは青色でハイライトがかかり，そのソースコード内のコードクローン数も表示される．ディレクトリツリーでファイルを選択すると，そのソースコードが右側のソースコードビューに表示される．コードクローンの部分はハイライトがかかって表示される．

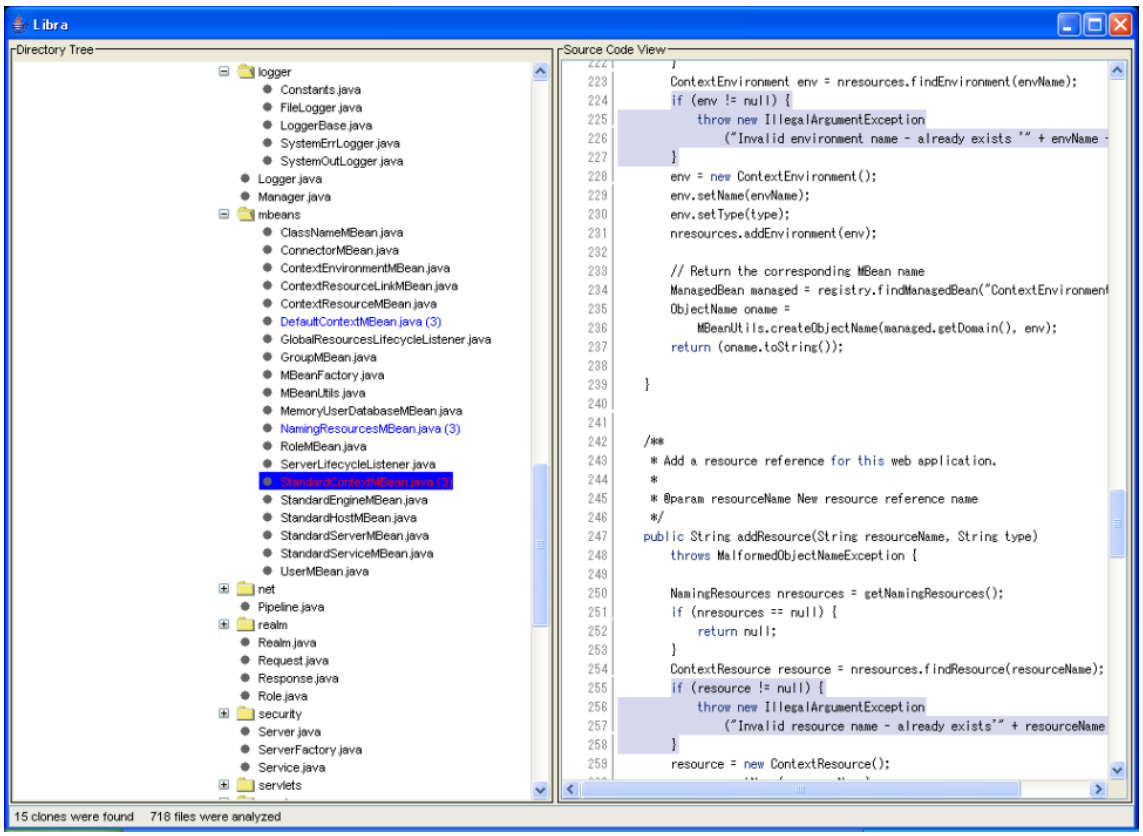


図 2-13　Libra 出力例

Libra の適用事例としては，文献[Izumida2003]と文献[Morisaki2008]のものがある．文献[Izumida2003]では,類似バグの修正漏れ対策として Libra の利用可能性を評価した．オープンソースのかな漢字変換ソフトウェア“かんな”の不具合修正への適用を行った．バージョン 3.6 と 3.6p1 の間で実施されたバッファオーバフローに関する修正が 21 カ所存在しており,そのうちの 1 カ所を検索コード片として,他の 20 カ所の検索を試みた．適合率　100%，再現率 81%という結果を得た．従来よく使われる grep を用いた検索結果では，適合率 34%，再現率 95%となっていた．

文献[Morisaki2008]では，コードクローン検索による類似不具合の検出の有効性を 3 つの商用ソフトウェア開発プロジェクトのデータを用いて実証的に評価した．適用対象は,Windows アプリケーション(言語:C++ ，サイズ:13.5k,修正箇所:16 カ所),Windows クライアントサーバアプリケーション（言語：Java ，サイズ：7.2k，修正箇所：10 カ所），UNIX サーバアプリケーション（言語：C++，サイズ：20.7k，修正箇所:10 カ所）であった．評価方法としては，先ず，修正履歴をもとに全ての修正箇所を修正前の状態(不具合が再現する状態）に戻す．次に，修正対象となったソースコード片を取り出し，クローンとなっている箇所を検索する．最後に，検索結果が類似不具合を正しく検索できているかどうかを確認した．結果として，Windows アプリケーションは適合率 91.8%，再現率 86.9%，Windows クライアントサーバアプリケーションは適合率 80.4%　再現率 85.4%，UNIX サーバアプリケーションは適合率 63.5%，再現率 90.0%であった．平均すると，78.6%の適合率と 87.4%の再現率が確認された．この結果に対して，現場の技術者か

らは，以下のような意見を得た．

・修正コードに関する記憶や知識があれば，より再現率が大きくなるような工夫ができる．

・派生製品や類似機能の不具合調査など迅速かつ網羅的な確認に特に有用である．

・今回は開発終了後(出荷後）のソースコードで結合，システムテスト以降の記録を対象としたが，コーディング，単体テスト実施等にプログラマが類似箇所を調査する際にも役立つ．

・類似不具合の検索を，grep 等のキーワード検索等の方法で検索キーを試行錯誤しながら検索した場合，1 件あたり数時間必要となることがある．本手法の場合，検索キーを試行錯誤しながら検索しても，数十分程度に短縮できそうである．

完全な網羅性は無いが，高い精度で類似バグ検出ができ，適用コストも低いという評価を得た．

2.1.3 研究目標

2.1.2 で述べた通り，研究グループは，これまでコードクローン分析に関する研究を一貫して続けてきている．その成果，経験を生かし，本研究で設定している以下の４つの課題についての研究目標・到達目標をまとめる．

(1) ソースコード理解支援

研究目標：細粒度でなるべく多くのコードクローンを高速に検出する手法を開発する．具体的には，これまでに開発されてきている幾つかのコードクローン検出ツールと比較して，冗長なコードクローンをなるべく含まず，処理が高速で，検出したコードクローンの再現率（検出したコードクローン数の検出すべき総コードクローンに対する割合)，適合率（検出したコードクローンが正しいコードクローンである割合）が高いコードクローン検出ツールを実現する．

到達目標：開発したコードクローン検出ツールについて，他のコードクローン検出ツールとの比較を，主に，オープンソースソフトウェア群を対象として行う．

(2) リファクタリング支援

研究目標：コードクローンに対して，より多くのリファクタリングパターンに対応できるツールの開発/集約方法のガイドラインの開発を行う．

到達目標：開発したコードクローン集約ツールを，主に，オープンソースソフトウェア群を対象として適用し，得られた結果に基づき，当該リファクタリングに関する集約ガイドラインをまとめる．

(3) 再利用ライブラリ作成支援

研究目標：大規模なソースコード群から，再利用に適切な大きさ（例えば，メソッド，関数単位）のコードクローンをなるべく高速に検出するための手法とツールの開発を行う．

到達目標：開発したコードクローン検出ツールを，主に，オープンソースソフトウェア群を対象として適用し，得られたコードクローンが再利用に有用かどうかを評価する．

(4) 違反流用コード発見支援：ライセンス違反等が疑われるコードの検出

研究目標：大規模なソースコード群から，流用として適切と考えられる大きさ（例えば，メソッド，関数単位）のコードクローンをなるべく高速に検出するための手法とツールの開発を行う．

到達目標：開発したコードクローン検出ツールを，主に，オープンソースソフトウェア群を対象として適用し，流用されたコード片が検出できるかどうかを評価する．

2.2 研究の活動実績・経緯

(1) 活動実績と経緯

研究の活動実績，経緯をまとめる．表 2-1 に，研究実施実績を示す．

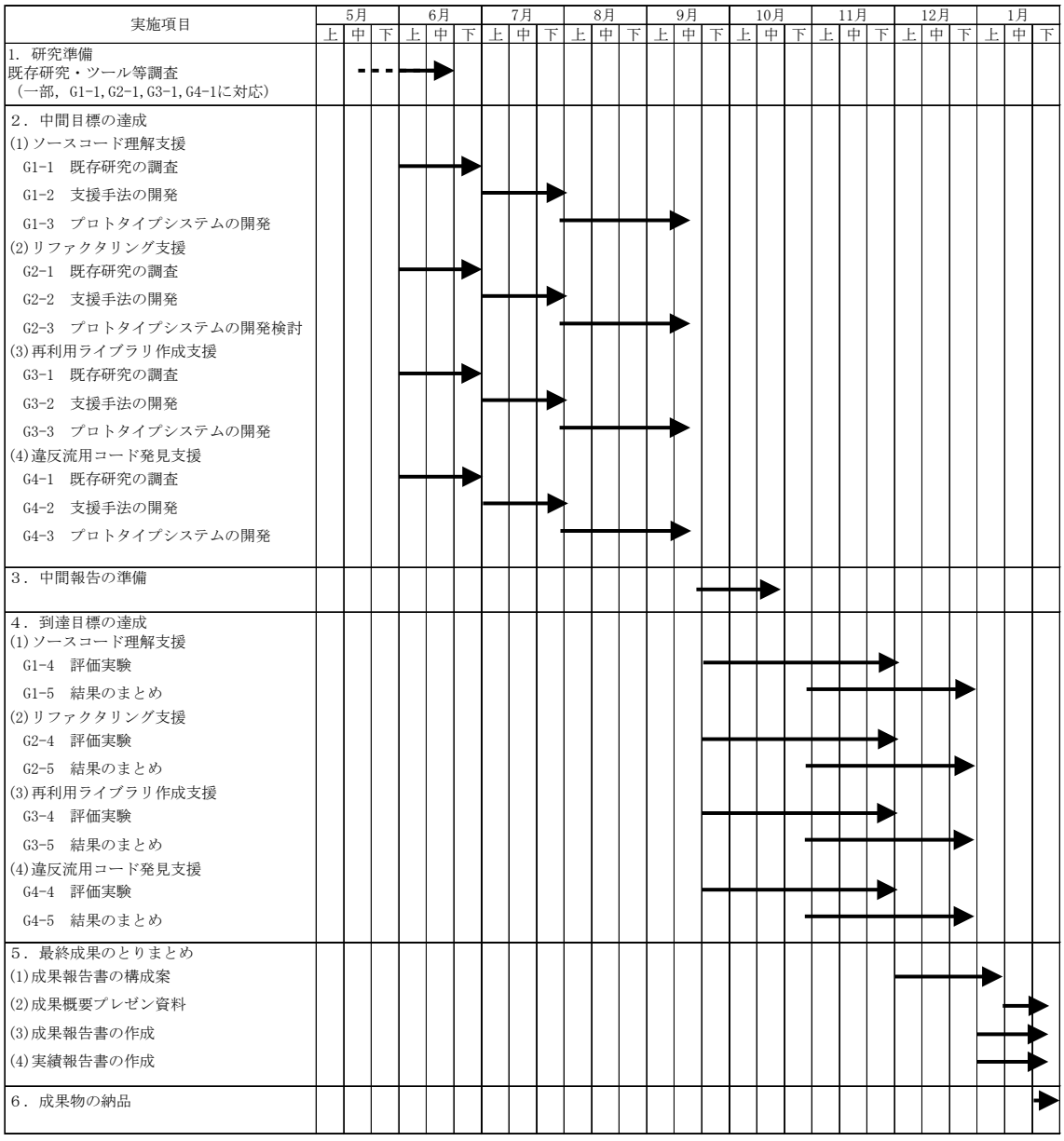
先ず，活動実績の概要を説明する．2012 年 5 月中旬～6 月中旬にかけて，研究準備として既存研究や・ツール等の調査を行った．6 月～9 月中旬は，4 つのテーマについての研究目標，すなわち，既存研究の調査，支援手法の開発，プロトタイプシステムの開発(一部，検討)を行った．9 月中旬～10 月末にかけては，中間報告の準備を行い，11 月 1 日に中間報告会を IPA/SEC にて実施した．4 つのテーマについては，引き続き到達目標の達成のため，9 月下旬～12 月下旬にかけて，評価実験，評価のまとめを行った．12 月以降は，最終成果のとりまとめを行い，成果報告書の構成案を 12 月下旬に提出し，2013 年 1 月 18 日に成果報告書のドラフトを提出し，1 月 24 日に最終成果報告会を IPA/SEC にて実施した．なお，最終成果報告書は 1 月 31 日に提出した．

実施期間中は，研究チームメンバーが参加する進捗報告会を隔週開催し，研究の進捗状況を確認すると共に，助言・指導を行った．また，必要に応じて，各研究チーム内でのミーティングも適宜実施した．内部レビューを委託期間中に 2 回実施した．内部レビューの手順は，以下の通りである．

1. 内部レビュー前に報告資料をメンバーに配布し，各メンバーが事前に問題点等を洗いだす．
2. メンバー全員で内部レビューを行い，1.の結果得られた問題点をまとめる．
3. 問題が指摘された箇所については担当メンバーが対策案を作成する．
4. 対策案の内容を研究責任者が確認後，担当メンバーが実施する．実施結果については，進捗報告会で報告する．

但し，進捗報告会の実施に当たっても事前に関連資料の送付を行い，また，研究メンバーが所属している研究室内のミーティング等でも内容確認を行っていたため，進捗報告会そのものが内部レビューとほぼ同じ役割を果たしていた．

表 2-1　研究実施実績



(2) 学会等参加状況とその成果

当該研究実施期間における学会等の参加状況とその成果をまとめる．

・6 月

6/28～7/1 で岡野浩三准教授が本研究の情報収集のために国際会議 ICEIS 2012（企業情報システムに関する国際会議）に参加した．ICEIS 2012 はエンタプライズ情報システムに関する国際会議であり，今回で 14 回目を数える．主にヨーロッパの企業大学研究者が集う．近年は ENASE　ソフトウェア工学における新アプローチに関する国際会議と共同で開催している．エンタプライズソフトウェアシステムを主対象に，モデル駆動開発やビジネスロジックの構築に関する論文を多く扱う国際会議であった．本

研究で対象としているソフトウェア保守に関する論文としては，再利用性の向上を念頭にモデル駆動開発のフレームワークの提案とその有効性評価実験の報告や要求記述の動的な変更に対する保守の問題をモデル駆動開発の技術で解決する枠組みの提案があった．エンタープライズシステムにおける現時点の研究のメインストリームとそれとの保守の研究課題の関係についての情報収集ができた．

・7 月

7/2 に楠本が，企業の実務家が集まる JFPUG FP 活用研究会で，企業の現場におけるソフトウェア生産管理・分析作業における取り組み内容や課題と本研究で対象としているコードクローン分析の関係に関する情報収集を行った．具体的には，ソフトウェア保守におけるコードクローンの問題や本研究で実施する内容について，簡単に紹介した．意見交換をした参加者は，開発の生産性分析におけるコードクローンの影響についての質問があり，生産管理データの収集項目としてコードクローンを取り上げるかどうかを検討したいということになった．今後，研究会において，開発現場・ユーザの立場から，研究成果に対するコメントをいただけるよう依頼した．

7/23 に松下准教授が，第 2 回 論文誌ジャーナル「ソフトウェア工学」特集号編集委員会に参加し，立命館大学の丸山教授らとソフトウェア工学，ソフトウェア保守に関して，意見交換及び情報収集を行った．ソースコード理解の研究においては，小規模な実験室レベルの評価ではなく，実際広く利用されているオープンソースソフトウェアを対象にした実践的な評価を積極的に行い，その有用性を主張することが重要であるという指摘を受けた．また，リファクタリング支援の研究に関しては，種々の手法の適用を単に考えるのではなく，リファクタリングを行う際のユースケースをまず適切に設定した上で，そのユースケースにおいて適切な支援方法を考案，あるいはガイドラインを作成し，そのうえで設定したユースケースに即した評価を行うことが重要であると指摘を受けた．

7/27,28 に岡野准教授が，電子情報通信学会ソフトウェアサイエンス/知能ソフトウェア工学合同研究会に出席し，ソフトウェア開発に関する最新の技術報告の収集をするとともに，研究者とソフトウェア開発に関する意見交換を行った．上流工程の要求仕様獲得や仕様分析に関する話題が多くあり，研究会参加者の名古屋大結縁教授，ＮＩＩ中島教授，芝浦工大松浦教授とソフトウェア要求仕様の形式化とソフトウェア保守との関係，保守工程の自動化手法について議論した．ソフトウェア機能のモジュール化の分類における保守作業の難易度の分類などの研究上のアイディアを得た．

・8 月

8/6 に楠本が，企業の実務家が集まる JFPUG FP 活用研究会で，企業の現場における生産性評価，プロセス改善，初期見積における取り組み内容や課題と本研究で対象としているコードクローン分析の関係に関する情報収集を行った．先月に引き続き，本研究で実施する内容と現状について，簡単に紹介した．具体的には，ソースコード理解支援，リファクタリング支援，再利用ライブラリ作成支援，違反流用コード発見支援，それぞれについて提案手法のキーアイデアを紹介した．参加者には興味は持って頂いたようであるが，それぞれの手法で得られる効果や実験結果等の具体的な話を今後聞きたいとコメントがあった．

8/27-29 に岡野准教授，肥後助教，石尾助教がソフトウェアエンジニアリングシンポジウム 2012 に出席し,コードクローンに関する最新の技術報告の収集をするとともに，研究者とソフトウェア開発に関する意見交換を行った．石尾助教は，リファクタリング支援技術に関する評価実験の方法を検討するため，特にリファクタリングの正しさを検証するためのソフトウェアテスト技術に関する情報収集に当たった．肥後助教は，違反コード検出技術の現状について，これまでに得られた成果について参加者と意見交換をすることで,今後の研究遂行に対するフィードバックを収集した.また，シンポジウム後，8 月 30 日に東京工業大学で実施されたソフトウェア工学に関する勉強会に参加し,ソフトウェア工学分野におけるトップカンファレンスである ICSE で発表された論文に関する情報を収集した．岡野准教授は，8/27 にワークショップ「形式手法の今と未来」に参加し，開発者が試作したソフトウェアの設計や実装の正しさを検証する方法について，参加者との議論を行った.

・9 月

8/31～9/10 で，岡野准教授が IWIN2012(ヨーロッパを中心とする世界のソフトウェア企業･研究者が集まる国際会議)で本研究テーマであるソフトウェア開発保守に関する視点からモデル指向開発の研究の情報収集を行った．あわせて，本研究に関する意見交換を会議のゲストスピーカーであるバーミンガム大学の Dr. Bordbar と行った．Dr. Bordbar が開発している自然語で書かれた保守用のコメントからの OCL で記述された形式仕様記述への変換技術が，コードクローンセットからソースコード理解の目的に合うものをフィルタリングするという方法に使えそうな印象を得た．実際には，OCL の導出の手前の自然言語処理技術を利用し，重要キーワード自動導出を行うことによりフィルタリングに活用する方式の検討，コード自動生成などへの応用が述べられた.

9/16～21 で，楠本が実証的ソフトウェア工学国際会議とそれに関連する会議（ESEM2012, ISERN2012）に出席し，意見交換・情報収集を行った．ソフトウェア保守におけるコードクローンの問題については，意見交換をした参加者は好意的な意見であった．特に，元請けの立場であることが多い大手ベンダからの参加者からは，再委託先でのコードの流用についての問題を認識しており，研究テーマの違反流用コード発見支援･ライセンス違反等が疑われるコードの検出に興味を持ってもらえた.また，一般的な意見として，現場での利用を考えると技術的な内容よりも，それを利用することでどれだけの効果があるかという評価の部分を重視すべきであるという意見を伺った．特に後者の意見については，提案手法の評価実験を計画する上で，考慮することとなった.

9/23～27 で肥後助教がソフトウェア保守国際会議(ICSM2012)と関連国際会議（SCAM2012）に出席し，意見交換・情報収集を行った．ソースコード中の繰り返し処理の部分が，コードクローン検出において悪影響を与えているとのことについて，意見交換した多数の方々から賛同を得た.特に SCAM はソースコード解析専門の国際会議であり，この分野に詳しい研究者が多数参加しているため，彼らから研究の着眼点について賛同を得ることができたのは大きいと考えられる．しかし，評価については慎重に行わなければならないとの意見を頂いた．コードクローン検出の精度を評価する

ためには，検出するべきコードクローンをどの程度実際に検出することができたのかを調査すべきである．しかし，検出すべきコードクローンの母集合を得ることは現実的ではないため，なにか他の方法を考慮する必要がある．

・10 月

10/15～18 で，肥後助教が国際会議 The Working Conference on Reverse Engineering (WCRE) に出席し，意見交換・情報収集を行った．多くのソフトウェア中に存在するコードクローンをライブラリ化の候補として抽出することについて，意見交換した多数の方々から賛同をえることができた．特に WCRE は，リバースエンジニアリング専門の国際会議であり，この分野に詳しい研究者が多数参加しているため，彼らから研究の着眼点について賛同を得ることができたのは大きいと考えられる．しかし，具体的な検出法は慎重に考える必要があるとの意見も頂いた．多数のソフトウェアからコードクローンを検出するためにハッシュを用いた高速な検出法とする予定であるが，そのやり方だと，エラー処理等に少し違いがあった場合でもコードクローンとして検出することができなくなってしまう．たとえエラー処理等が異なっていてもプログラムの処理として同一であれば検出できる手法にすることが望ましいとの意見を頂いた．

10/10～12 で，土屋教授が東京ビックサイトで開催された ITPro Expo 2012 に出席し，意見交換・情報収集を行った．

・11 月

松下准教授が，11/1,2 に広島市立大学で開催されたソフトウェア工学研究会に出席し，本研究成果について参加者と意見交換を行った．また，土屋教授が，11/14～19 に新潟市で開催された国際会議 PRDC2012 に出席し,本研究成果について参加者と意見交換を行った．

・12 月

12/4～7 で，松下准教授，肥後助教が香港で開催された国際会議 APSEC2012 に出席し，本研究成果について参加者と意見交換を行った．複数プロジェクトからのライブラリ化を目的としたクローン検出および繰り返し部分を考慮したクローン検出に関して，その評価方法に関する情報収集とリファクタリング支援に関する評価方法を検討するために，形式手法を用いた検査に関する調査を行った．評価方法について，頂いた意見で多かったのは，既存の研究で実験対象となっているソフトウェア群をこの研究でも実験対象にすべきではないか，というものであった．既存研究と同じ実験対象に対して評価を行うことで既存研究との差をより正確に客観的に評価できるため，極力同じソフトウェア群を使うべきとの意見を得た．

12/13～15 で，岡野准教授，石尾助教が湯布院で開催された第 19 回 ソフトウェア工学の基礎ワークショップ（FOSE2012）に出席し，本研究成果について参加者と意見交換を行った．議論の結果として，我々の研究で実装した情報の提示方法と，考え方やツールの作り方に大きな差はなく，現行の我々の実装は，現在用いられている技術においては一般的なものであり，実験等にあたっても問題を生じないものであるとの結論を得た．

12/16～18 で，土屋教授がシンガポールで開催された国際会議 ICPADS に出席し，本研究成果について参加者と意見交換を行った．ソフトウェアを停止させず要求の変化

に合わせてソフトウェアコンポーネントを入れ替えるという研究では，結果として類似のコンポーネントを数多く用意しておく必要があり，我々の成果の応用が期待される．また， Code Reuse 攻撃を扱った研究では，脆弱性を含むコードパターンの存在が示唆されており，このような脆弱なコードの検出についても我々の成果の応用が期待される．

2.3 研究実施体制

研究実施体制を図 2-14 に示す．４つのテーマの実施について，(R1)ソースコード理解支援，(R2)リファクタリング支援，(R3)再利用ライブラリ・違反コード検出支援，の３グループを構成し，研究を行った．

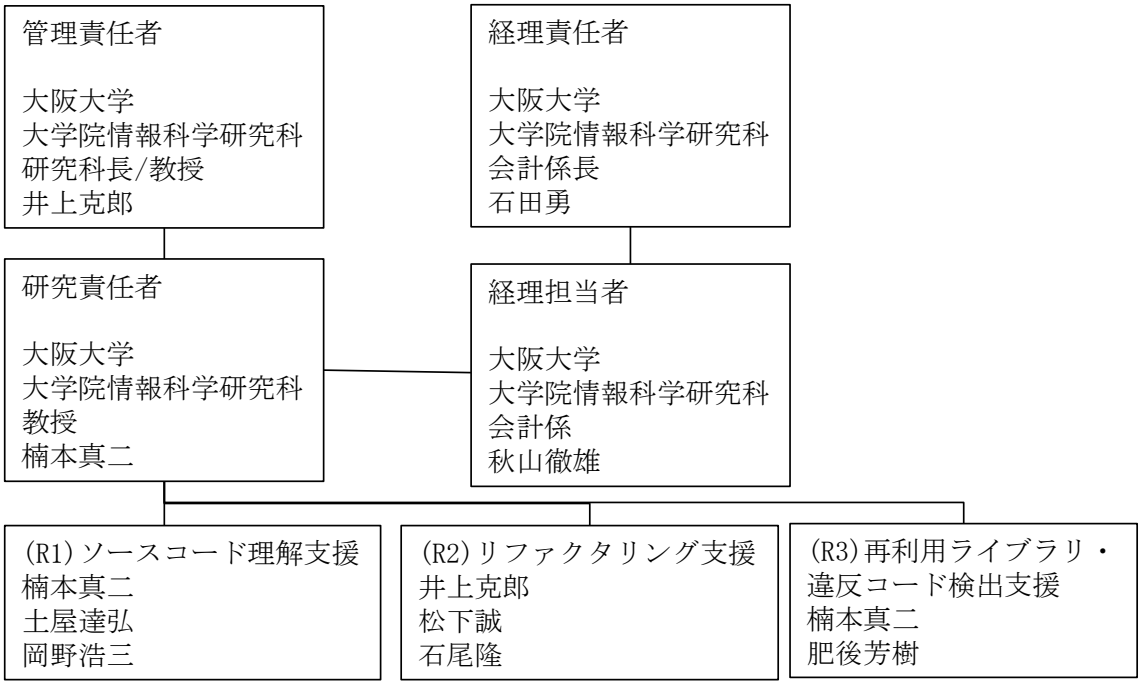


図 2-14　研究実施体制

研究グループの各メンバーのプロフィールは以下の通りである．

楠本真二
昭 63 年大阪大学基礎工学部卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部助手．平成 8 年同講師．平成 11 年同助教授．平成 14 年同大学大学院情報科学研究科助教授．平成 17 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．電子情報通信学会，情報処理学会，IEEE，IFPUG，PM 各会員．

井上克郎
昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59～61 年ハワイ大学マノア校情報工学科助教

授．平成元年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．平成 14 年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教授．平成 20 年国立情報学研究所客員教授．同年情報処理学会フェロー．同年電子情報通信学会フェロー．工学博士．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．

土屋達弘
平成 7 年大阪大学大学院博士前期課程修了，平成 8 年同大学院博士後期課程中退．同年同大学基礎工学部助手．平成 12 年同講師．平成 14 年同大学大学院情報科学研究科助教授，平成 24 同教授．平成 18 年～19 年スイス連邦工科大学ローザンヌ校客員研究員．コンピュータ工学．特に，分散アルゴリズム，モデル検査，ソフトウェアテスト，信頼性評価・最適化に関する研究に従事．電子情報通信学会，情報処理学会，IEEE，ACM 各会員．

岡野浩三
1990 年 大阪大学基礎工学部情報工学科卒業．1993 年 同大大学院博士後期課程中退．同大助手， 講師等を経て， 現在，大阪大学大大学院情報科学研究科准教授．博士(工学)．形式手法によるソフトウェア開発・検証方法などの研究に従事．情報処理学会，電子情報通信学会，IEEE_CS 各会員

松下誠
平成 5 年大阪大学基礎工学部情報工学科卒業．平成 10 年同大学大学院博士後期課程修了．同年同大学基礎工学部情報工学科助手．平成 14 年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助手．平成 17 年同専攻助教授．平成 19 年同専攻准教授．博士（工学)．ソフトウェア開発環境，リポジトリマイニングの研究に従事．日本ソフトウェア科学会，ACM 各会員

肥後芳樹
平成 14 年大阪大学基礎工学部情報科学科中退．平成 18 年同大学大学院博士後期課程修了．平成 19 年同大学院情報科学研究科コンピュータサイエンス専攻助教．博士（情報科学)．ソースコード分析，特にコードクローン分析やリファクタリング支援に関する研究に従事．情報処理学会，電子情報通信学会，日本ソフトウェア科学会，IEEE 各会員．

石尾隆
平成 15 年大阪大学大学院基礎工学研究科博士前期課程修了．平成 18 年同大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD)．同年ブリティッシュコロンビア大学ポストドクトラルフェロー．平成 19 年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助教．博士（情報科学)．データフロー解析，アスペクト指向プログラミングに関する研究に従事．日本ソフトウェア科学会，ACM，IEEE 各会員．

また，(R1)～(R3)それぞれについて，関連研究論文調査・既存ツール調査，プロトタイ

プシステムの実装・修正，テスト，評価実験等の作業補助として，学生アルバイトを雇用した．

３．研究成果

3.1 研究目標１「ソースコード理解支援」

3.1.1 当初の想定

(1) 想定する仮説等

本テーマの研究目標は，これまでに開発されてきている幾つかのコードクローン検出ツールと比較して，冗長なコードクローンをなるべく含まず，処理が高速で，検出したコードクローンの再現率（検出したコードクローン数の検出すべき総コードクローンに対する割合)，適合率（検出したコードクローンが正しいコードクローンである割合）が高いコードクローン検出ツールを実現することである．

本テーマを実施するに当たって，想定する仮説は,「冗長なコードクローンをなるべく含まないように検出するため冗長なコードクローンをうまく特定できるか？」と「特定したコードクローンを高速に検出できる手法・プロトタイプシステムを開発できるか？」の 2 点になる．

(2) 当初の到達目標

上述した仮説を確認するための，到達目標としては，どのようなコードクローンを冗長であると見なすかと言うことと，冗長でないコードクローンを高速に，かつ，再現率・適合率を高く検出できるプロトタイプシステムの開発を行うということになる．

(3) 当初の期待される効果

上記の到達目標が達成されることで，評価実験とその有用性評価を実施することが可能となる．

3.1.2 研究プロセスと成果

(1) 研究プロセス

冗長なコードクローンの定義，冗長なコードクローンの検出方法の提案，提案手法を実現したプロトタイプシステムの開発，評価実験の順に実施する．

(2) 具体的な研究成果の内容

① 冗長なコードクローン

コードクローンは，その検出手法によって定義が異なるため，先ず，対象とするコードクローン検出手法を決める必要がある．コードクローン検出手法は，コードクローンをどの単位で検出するかによって，大きく以下の５つに分類できる(図 3-1-1)[Higo2008]．

・行単位

・字句単位
・抽象構文木
・プログラム依存グラフ
・メトリクス

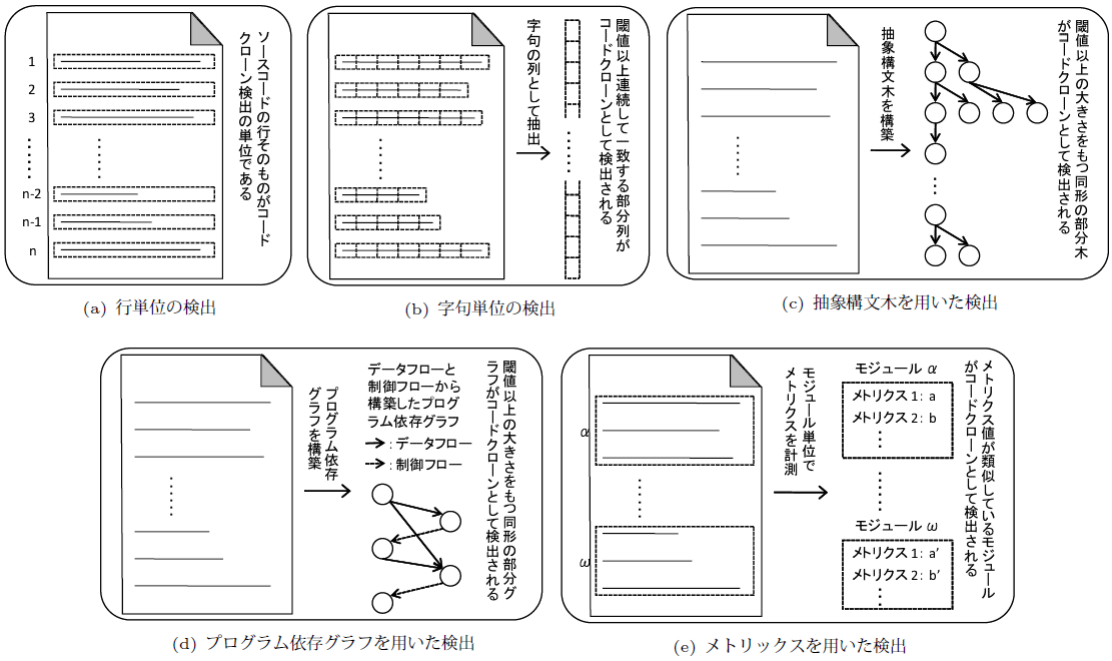


図 3-1-1　コードクローン検出手法

これらの中でも広く利用されているのは，行単位や字句単位の検出手法である．その理由を下記に示す．

・行単位や字句単位の検出手法は，ソースコードの字句解析と簡単な構文解析を行うのみであり，抽象構文木やプログラム依存グラフ等を生成しない．よって高速に検出処理を行うことが可能であり，大規模ソフトウェア，多数のソフトウェア間，ソフトウェアのリビジョン群に対しても適用可能である．
・ソースコードの深い解析を必要としないため，他の検出手法に比べて複数のプログラミング言語に対応させるのが難しくない．たとえば，プログラム依存グラフを用いた検出手法は，プログラム依存グラフを構築するための解析を行わなければならない．実際に，代表的な行単位や字句単位の検出ツールである Simian[Simian]や CCFinder[Kamiya2002] は，C/C++，Java，COBOL 等の社会で広く利用されている複数のプログラミング言語に対応している．

本研究では，コードクローン検出の処理が高速であるということを一つの目標としているため，対象として字句単位の検出手法を対象とする．

既存の行単位や字句単位の検出手法には，ソースコード中の同じ命令が繰り返し記述された部分（以降，繰り返し部分と呼ぶ）において，冗長なコードクローンを検出する，並びに検出すべきであるコードクローンを検出できない，という課題が指摘されている．図

3-1-2 に一例を示す. 図 3-1-2(a)と図 3-1-2 (b)の switch 文は, それぞれ 5 つと 3 つの case エントリを含んでいる. これら 8 つの case エントリは, リテラルが違うのみの繰り返し構造となっている. この switch 文に対して既存の検出ツールを適用すると 6 つものコードクローンのペアを検出してしまう. しかし, このソースコードにおいて, ユーザが必要とする情報は, 図 3-1-2 (a)と図 3-1-2 (b)の switch 文は共に StringBuffer を用いた文字列の追加処理を行なっているということであろう. よって, switch 文全体をコードクローンのペアとして検出することが望ましい.

そこで, 本テーマでは, 冗長なコードクローンを繰り返し部分を含むコードクローンと考え, より有益なコードクローン検出結果を得るために, ソースコード上の繰り返し構造を折りたたむという検出の前処理を行った上でコードクローン検出する手法を提案する. 繰り返し部分を折りたたむことで, 着目する必要のない冗長なコードクローンの検出を抑止するとともに, 把握すべきコードクローンが見つかることが期待される.

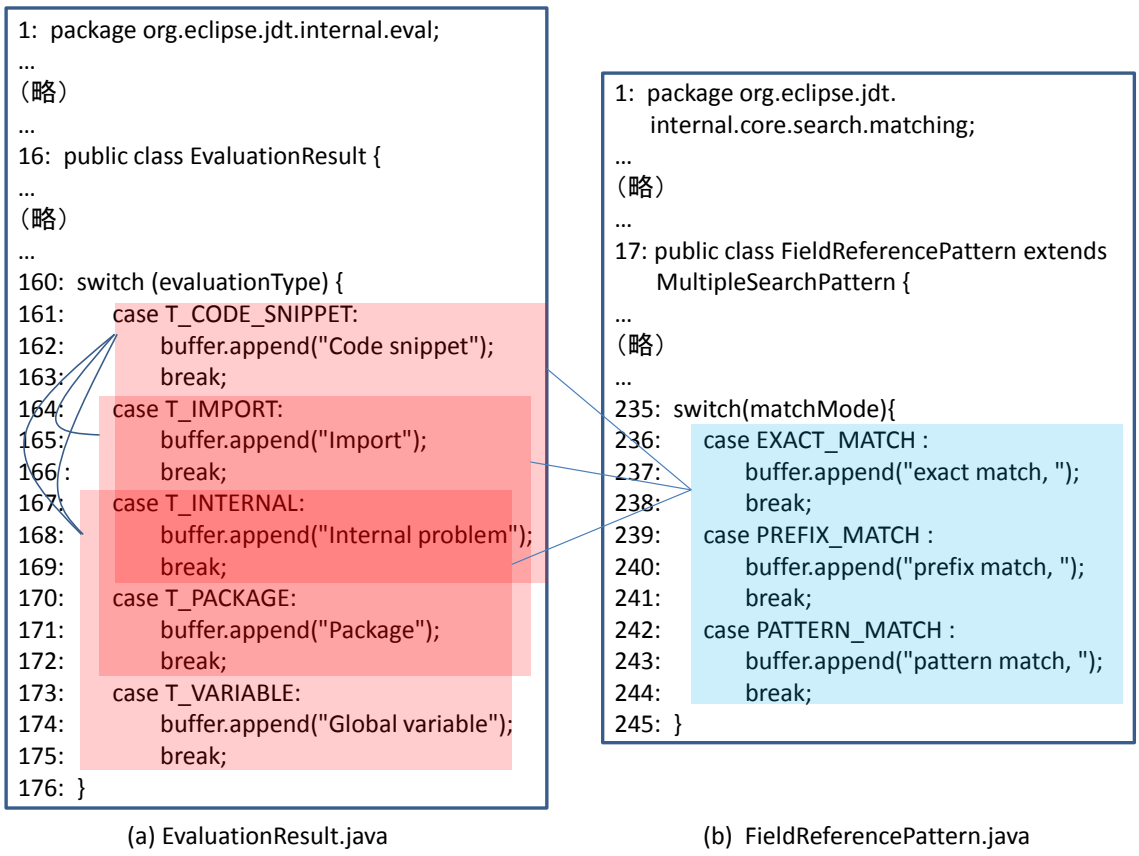


図 3-1-2　繰り返し部分におけるコードクローン検出例

② 提案手法

ここでは, 提案手法である繰り返し部分の折りたたみについて, ①で挙げた例を元に説明する. この例は switch 文の各 case エントリが繰り返されている. この繰り返し部分を折りたたむと図 3-1-3 のように 2 つの switch 文は 1 つの case エントリのみをもつ構造になる. 繰り返し部分の繰り返し回数を記憶し, また, ユーザ定義名を特別な同一の字句に

置き換えると，単純な字句単位の解析で switch 文全体をコードクローンとして検出することができる．検出例を図 3-1-4 に示す．

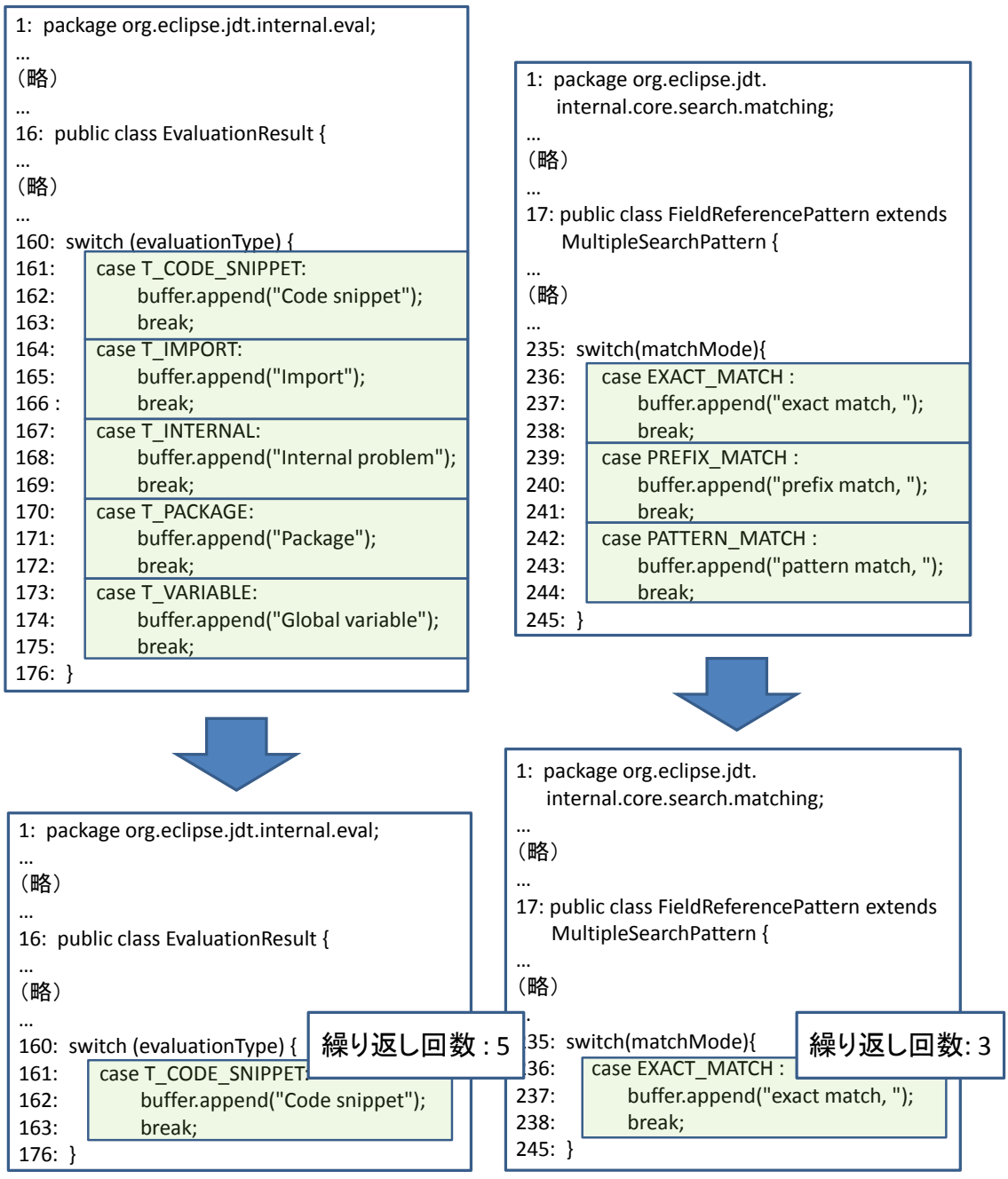
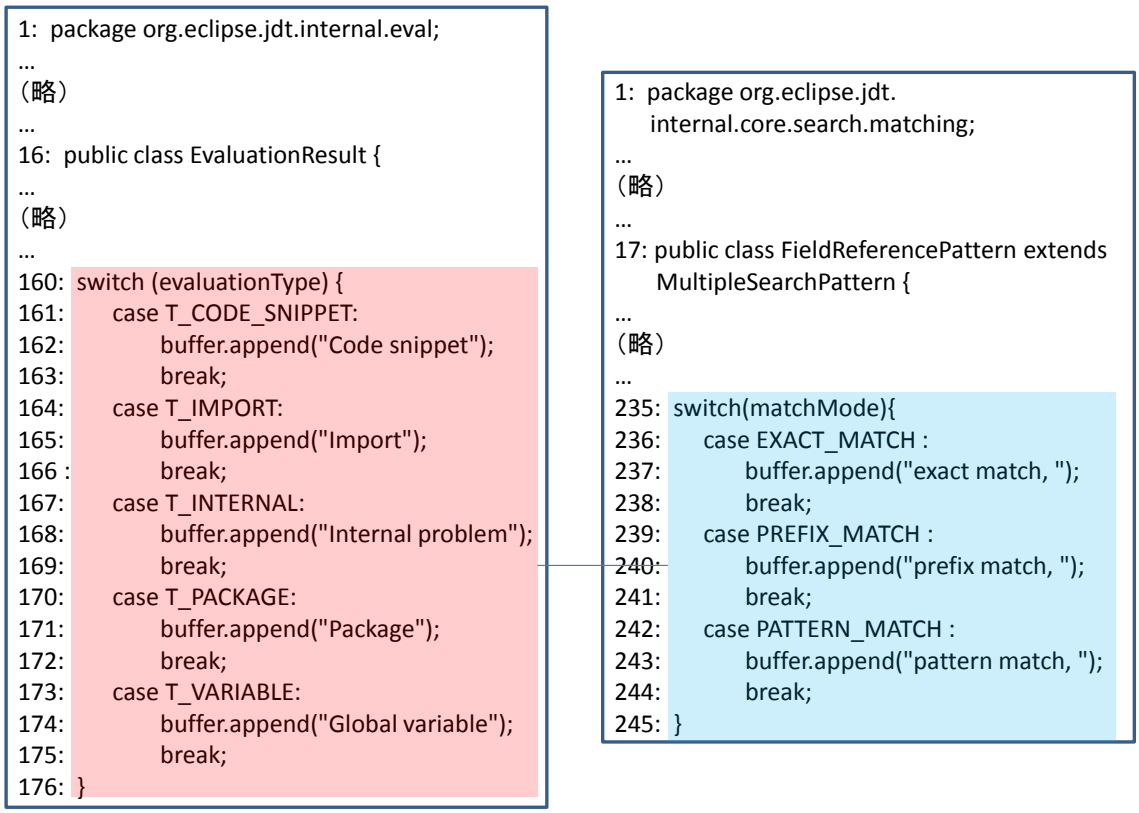


図 3-1-3　繰り返し部分の折りたたみ例



(a) EvaluationResult.java　　(b) FieldReferencePattern.java

図 3-1-4　繰り返し部分の検出例

折りたたみのアルゴリズムを図 3-1-5 に示す．引数の str は折りたたむ文字列，repeatCount は折りたたむ文の数を表す．例えば，repeatCount が 2 であれば「文 1，文 1」と「文 1，文 2，文 1，文 2」といった繰り返しを折りたたみ，「文 1，文 2，文 3，文 1，文 2，文 3」のような繰り返しは折りたたまない．アルゴリズム中の isSentenceEnd()は，与えられた文字が";"，"{"，"}"のいずれかであれば true を，そうでなければ false を返す手続きである．また，length()は，与えられた文字列の長さを返す手続きである．6-18 行目と 21-30 行目で隣接する 2 つの文を取得する．31-37 行目で隣接する 2 つの文が同じだったときの折りたたみの処理を行う．repeatCount の数だけ文の折りたたみを行うと，最後に 40 行目で折りたたまれた文字列を返す．

③ 提案手法のプロトタイプシステム

1）概要

プロトタイプシステムは Java を用いて実装した．現在のところ，コードクローンの検出が可能な言語は Java と C である．提案手法を組み込んだコードクローン検出は，検出対象のソースコードを入力とし，コードクローンの情報を出力とする..

```
Input: str, repeatCount(≧ 1)
Output: str after folded
 1: strlen ← length(str)
 2: for i = 0 to repeatCount do
 3:    left ← 0
 4:    loop
 5:       flg ← true; index ← left; tmpleft ← left; count ← 0;
 6:       while count ≦ i and index < strlen do
 7:          if isSentenceEnd(str[index]) then
 8:             if flg then
 9:                k ← index + 1; flg ← false
10:             end if
11:             count ← count + 1
12:          end if
13:          index ← index + 1
14:       end while
15:       if index > strlen then
16:          break
17:       end if
18:       tmp ← str[left..index − 1]
19:       count ← 0
20:       left ← index
21:       while count ≦ i and index < strlen do
22:          if isSentenceEnd(str[index]) then
23:             count ← count + 1
24:          end if
25:          index ← index + 1
26:       end while
27:       if index > strlen then
28:          break
29:       end if
30:       tmp2 ← str[left..index − 1]
31:       if tmp = tmp2 then
32:          str ← str[0..left − 1] + str[index..strlen]
33:          strlen ← length(str)
34:          left ← tmpleft
35:       else
36:          left ← k
37:       end if
38:    end loop
39: end for
40: return str
```

図 3-1-5　折りたたみのアルゴリズム

2）処理の流れ

提案手法の全体の流れを図 3-1-6 に示す．コードクローン検出は以下の 5 ステップで行われる．

ステップ 1：字句解析・正規化

字句解析によって入力ソースコードをトークン列に変換する．この際，コメントや改行記号，タブ，空白は取り除かれる．次に，ユーザ定義名を"$"に変換する．同時に，中括弧のネストを認識することで，メソッドや関数の境界を識別し，トークン列内におけるそれらの境界に"#"を埋め込む．

ステップ 2：変形処理

このステップでは字句解析・正規化によって得られた文字列を変形ルールに従って変形する．Java と C の変形ルールを以下に示す．

・パッケージ名を取り除く

PackageName + '.' + ClassName → ClassName

パッケージ内のクラスやインターフェースを呼び出す命令があった場合，パッケージ名を省略する．なお，この変形ルールは Java にのみ適応する．

・テーブルの初期化を取り除く

'=' '{' InitalizationList '}' → '=' '{' '}'

テーブルを初期化するときに要素が指定されていれば，それを取り除く

ステップ 3：繰り返し部分の折りたたみ

変形処理によって得られた文字列の中で繰り返している部分を取り除く．まず，";","{","}"を「区切り文字」と定義し，区切り文字で終わる文字列を「文」と定義する．次に，変形処理で得られた文字列を文単位で区切る．隣接する文が同じであれば同じ文が繰り返されているということであり，繰り返されている後ろの文を取り除く．また，複数の文の繰り返し（「文 1，文 2，文 1，文 2」のような繰り返し）を含む文字列に対しても同様の処理を行う．

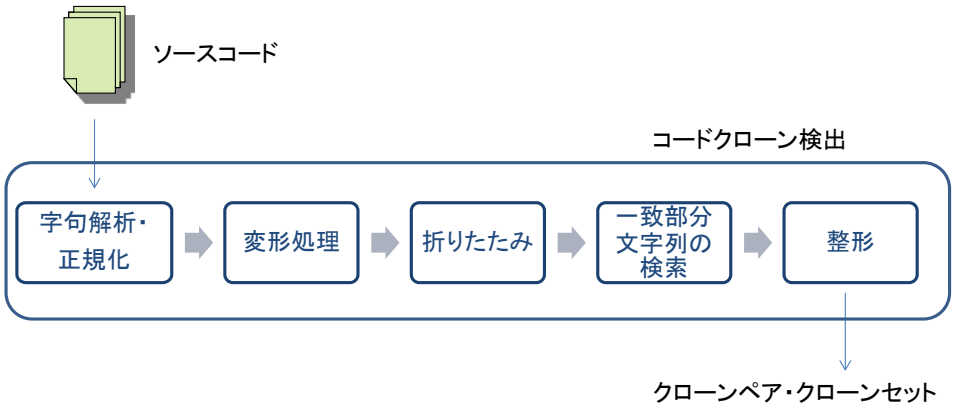


図 3-1-6　全体の流れ

3）評価実験

以下の 2 つの実験を行う．

実験 A：折りたたみの有無による精度比較実験
実験 B：提案手法と既存手法の精度比較実験

実験では，最小コードクローン長を 30，繰り返しの最大値を 5 の閾値を用いる．
実験では，Bellon らの実験で使用されたデータ[Bellon2007]を検出すべきコードクローン群とする．このコードクローン群は，8 つの対象ソフトウェアから複数のコードクローン検出ツールを用いて検出し，さらに検出結果から Bellon らの目視確認によって抽出されたコードクローンの集合である．表 3-1-1 に対象ソフトウェアの概要を示す．また，ツールが検出したクローンペアの集合をクローン候補，Bellon が抽出したクローンペアの集合を正解クローンと呼ぶ．しかし，正解クローンの中にコードクローンとして不適切なものが見られた．例えば，複数のメソッドにまたがってコードクローンとしている例や，メソッドがないクラスのみをコードクローンとしている例である．今回の実験では，正解クローンの中からこれらのコードクローンを取り除いたものを新たな正解クローン（以降，新正解クローンと呼ぶ）とした．

表 3-1-1　対象ソフトウェア

ソフトウェア名	言語	総行数	正解クローン数
netbeans	Java	14,360	55
ant	Java	34,744	30
jdtcore	Java	147,634	1,345
swing	Java	204,037	777
weltab	C	11,460	275
cook	C	70,008	440
snns	C	93,867	1,036
postgresql	C	201,686	555

図 3-1-7 に折りたたみの有無によるクローン候補数を示す．折りたたみにより検出されたクローン数が削減されていることが分かる．

次に，クローン候補が正解クローンにどれだけ類似しているかを good 値と ok 値[Bellon2007]を用いて判断し，再現率と適合率を算出する．再現率(recall)は，新正解クローン数に対する検出した中で新正解クローンであったものの数の割合である．適合率(precision)は，検出数に対する検出した中で新正解クローンであったものの数の割合である．本実験では，good 値と ok 値の閾値として 0.7 を用いた．なお，対象は，netbeans，jdtcore，wltab，postgresql である．

recall と precision についての実験結果を図 3-1-8 に示す．繰り返し部分を折りたたむことですべてのソフトウェアについて recall は減少し，precision は上昇した．

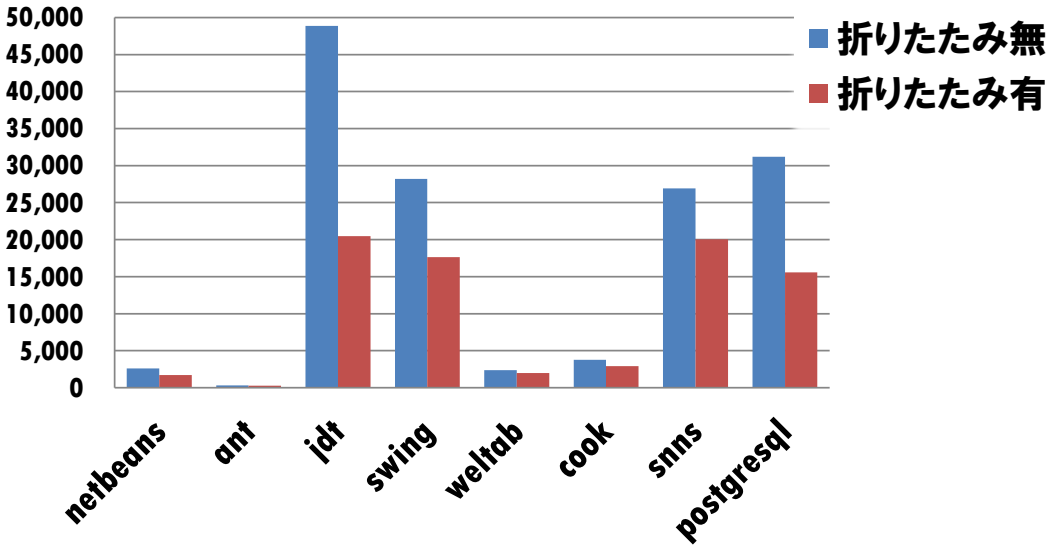


図 3-1-7　クローン検出数（折りたたみ有と無）

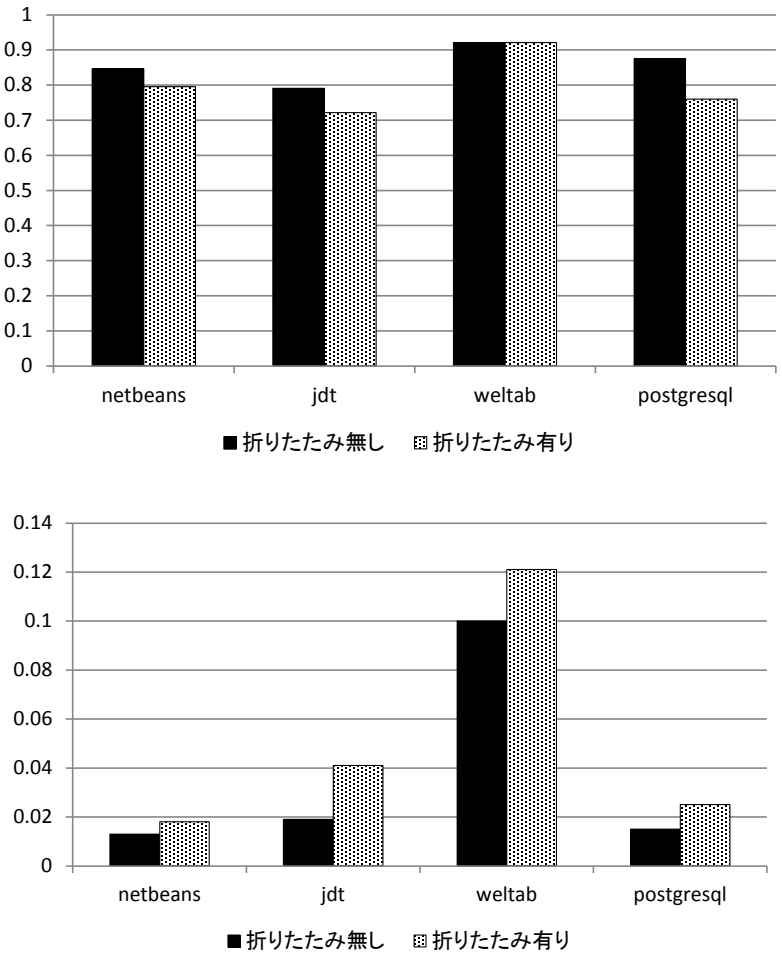


図 3-1-8　再現率（上）と適合率（下）

4)折りたたみを用いた検出例

提案手法による繰り返し部分の検出結果の 1 つを図 3-1-9 に示す．繰り返し部分において，各 if-else 節を別々にコードクローンとして検出することなく，if-else 節全体で検出できている．つまり，繰り返し部分をまとめて検出するのに成功し，提案手法の有効性を示すことができた．

```
1:  package org.eclipse.jdt.internal.core.jdom;
…
（略）
…
21: class DOMMethod extends DOMMember
      implements IDOMMethod {
…
（略）
…
469:  protected void offset(int offset) {
470:     super.offset(offset);
471:     offsetRange(fBodyRange, offset);
472:     offsetRange(fExceptionRange, offset);
473:     offsetRange(fParameterRange, offset);
474:     offsetRange(fReturnTypeRange, offset); }
```

(a) DOMMethod.java

```
1:  package org.eclipse.jdt.internal.core.jdom;
…
（略）
…
25: class DOMType extends DOMMember
      implements IDOMType {
…
（略）
…
483:  protected void offset(int offset) {
484:     super.offset(offset);
485:     offsetRange(fCloseBodyRange, offset);
486:     offsetRange(fExtendsRange, offset);
487:     offsetRange(fImplementsRange, offset);
488:     offsetRange(fInterfacesRange, offset);
489:     offsetRange(fOpenBodyRange, offset);
490:     offsetRange(fSuperclassRange, offset);
491:     offsetRange(fTypeRange, offset); }
```

(b) DOMType.java

図 3-1-9　提案手法による検出例

5）折りたたみによる検出数の減少

jdt との比較において，検出数が約 490,000 から 20,000 と半数以下に減少した．jdt のソースコードを調査したところ，jdt のソースコード中に if 文，if-else 節，switch 文内の case エントリ，try-catch 節などの繰り返しが非常に多く見られた．このような繰り返し部分に対し，提案する折りたたみ機能が効果的に働いたため，検出数の大幅な減少につながったといえる．jdt のプログラムにおける if 文の繰り返し例を図 3-1-10 に示す．また，jdt の検出結果のうち正解クローンであったものの数は 918 個から 836 個に減少した．これは，折りたたみによって繰り返し部分内のクローンを検出しなくなったためである（図 3-1-11）．繰り返し部分内のコードクローンは上述したように検出する利点がないので，本手法によるクローンの検出数の減少は妥当である．

```
1:  package org.eclipse.jdt.internal.core.jdom;
…
（略）
…
21: class DOMMethod extends DOMMember
      implements IDOMMethod {
…
（略）
…
469:  protected void offset(int offset) {
470:     super.offset(offset);
471:     offsetRange(fBodyRange, offset);
472:     offsetRange(fExceptionRange, offset);
473:     offsetRange(fParameterRange, offset);
474:     offsetRange(fReturnTypeRange, offset); }
```

(a) DOMMethod.java

```
1:  package org.eclipse.jdt.internal.core.jdom;
…
（略）
…
25: class DOMType extends DOMMember
      implements IDOMType {
…
（略）
…
483:  protected void offset(int offset) {
484:     super.offset(offset);
485:     offsetRange(fCloseBodyRange, offset);
486:     offsetRange(fExtendsRange, offset);
487:     offsetRange(fImplementsRange, offset);
488:     offsetRange(fInterfacesRange, offset);
489:     offsetRange(fOpenBodyRange, offset);
490:     offsetRange(fSuperclassRange, offset);
491:     offsetRange(fTypeRange, offset); }
```

(b) DOMType.java

図 3-1-10　jdt プログラム中の if 文繰り返し

```
13:  package org.netbeans.modules.javadoc.comments;
…
（略）
…
37:  public class JavaDocEditorPanel extends
javax.swing.JPanel implements
EnhancedCustomPropertyEditor {
…
（略）
…
637:  switch ( ks.getKeyCode() ) {
638:       case  KeyEvent.VK_B:
639:            boldButton.doClick();
640:            e.consume();
641:            break;
642:       case  KeyEvent.VK_I:
643:            italicButton.doClick();
644:            e.consume();
645:            break;
646:       case  KeyEvent.VK_U:
647:            underlineButton.doClick();
648:            e.consume();
649:            break;
650:       case  KeyEvent.VK_C:
651:            codeButton.doClick();
652:            e.consume();
653:            break;
654:       case  KeyEvent.VK_P:
655:            preButton.doClick();
656:            e.consume();
657:            break;
658:       case  KeyEvent.VK_L:
659:            linkButton.doClick();
660:            e.consume();
661:            break; } } }
```

JavaDocEditorPanel.java

図 3-1-11　繰り返し部分内のコードクローンの例

6) IT Spiral 実プロジェクトデータへの適用

開発したプロトタイプシステムを IT Spiral 実プロジェクトデータに適用した．適用結果を，表 3-1-2 に示す．結果として，折りたたみを行うことで，検出コードクローン数は約 2/3 に削減した．繰り返し部分を折りたたむことで検出できたコードクローン例を図 3-1-12 と図 3-1-13 に示す．図 3-1-12 は，assertEquals()文の繰り返し回数が異なる部分であり，図 3-1-13 は catch 節の繰り返し回数が異なるものになっている．

表 3-1-2　IT Spiral 実プロジェクトデータへの適用結果

	折りたたみ前	折りたたみ後
コードクローン検出数	8042	5272
実行時間	1.6秒	1.5秒

```
                              ⋮
1614: HibernateUtil.beginTransaction();
1615:
1616: JugyouKamoku jugyouKamoku = dao.load(102);
1617: assertEquals("授業コード:", "00000111", jugyouKamoku.getJugyouCode());
1618: assertEquals("授業形態:", JugyouKeitai.ENSYUU, jugyouKamoku.getJugyouKeitai());
1619: assertEquals("完了ステータス:", false, jugyouKamoku.isKanryouStatus());
1620: assertEquals("公開フラグ:", false, jugyouKamoku.isKoukaiFlag());
1621: assertEquals("年度:", 2008, jugyouKamoku.getNendo());
1622: assertEquals("単位:", 4, jugyouKamoku.getTani());
1623: assertNull(jugyouKamoku.getJugyouShousai());
                              ⋮
```

JugyouKamokuDAOTest.java

```
                              ⋮
105: DatabaseOperation.INSERT.execute(getConnection(),
106:     getDataSet("Kamoku_testLoad_init.xml"));
107:
108: Kamoku kamoku = service.load(5001);
109: assertEquals("授業コード", "00005001", kamoku.getJugyouCode());
110: assertEquals("科目名", "テスト", kamoku.getKamokuName());
111: assertEquals("単位", 2, kamoku.getTani());
112: assertEquals("登録年度", 2007, kamoku.getTourokuNendo());
113: assertEquals("削除フラグ", false, kamoku.isSakujoFlag());
114:
115: HibernateUtil.rollback();
                              ⋮
```

IKamokuServiceTest.java

図 3-1-12　検出例 1（IT Spiral 実プロジェクトデータ）

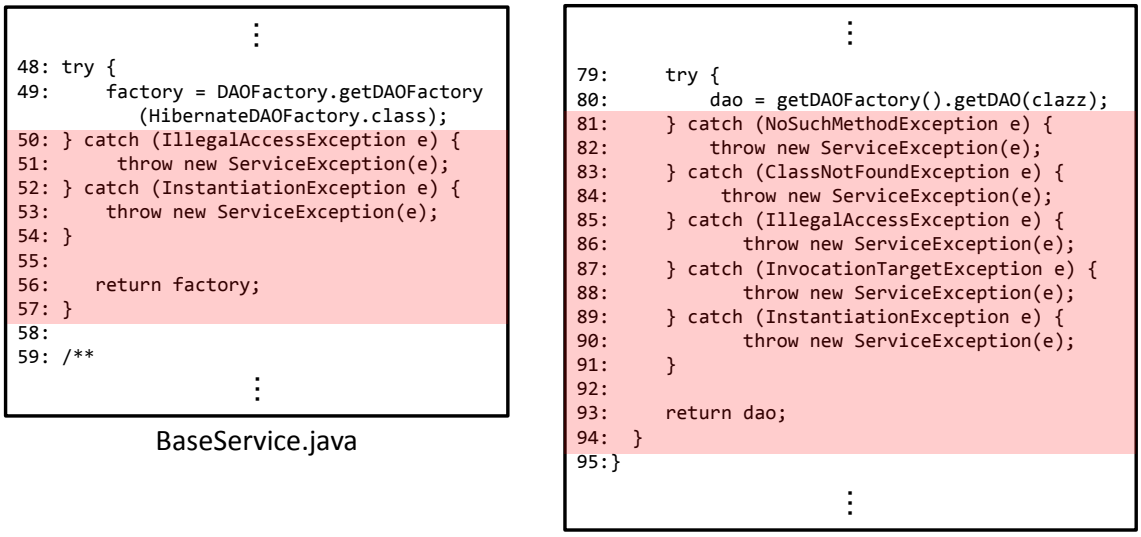


図 3-1-13　検出例 2（IT Spiral 実プロジェクトデータ）

7）検出速度の比較

表 3-1-3 に，3)で述べたオープンソースソフトウェアに対して，開発したプロトタイプシステムと CCFinder を適用した時のコードクローン検出時間の比較結果を示す．表 3-1-3 に示すとおり,開発したプロトタイプシステムの検出時間は CCFinder より短いことが分かる．

表 3-1-3　検出速度比較

	プロトタイプ	CCFinder
netbeans	1.63	1.99
ant	2.53	2.42
jdt	5.95	7.71
swing	6.88	8.63
weltab	3.37	2.01
cook	2.87	3.78
snns	3.66	6.22
postgresql	10.33	9.98
平均	4.65	5.34

3.1.3 実用化へ向けた課題と問題点

(1) 課題と問題点

本テーマでは，ソースコード中の繰り返し部分を折りたたんでコードクローンを検出する手法を提案した．提案手法を用いたコードクローン検出ツールを実装し，オープンソースソフトウェアに対して検出を行った．その結果，折りたたみ有りの場合と折りたたみ無しの場合で検出数が減少したことを確認した．さらに，さらに既存の検出手法との比較を通じて，検出されたコードクローンの評価を行った．その結果，繰り返し部分の折りたたみにより，既存手法の問題点を解消したことを示した．

今後の課題としては，先ず，検出対象の言語を増やすことが考えられる．また，インクリメンタルな検出に対応する必要がある．インクリメンタルな検出とは，検出処理で得た情報をデータベースに登録し，それ以降の検出処理においてデータベースを活用することである．これにより，検出時間を短縮できる．また，現在は，コードクローンの情報をテキストファイルに出力している．より実用性を高めるためには，表示方法の改良やEclipse プラグインとして実装するなど，ユーザインターフェースの充実も必要であると考えられる．

(2) 将来の応用方法

ソースコード中の繰り返し部分を折りたたんでコードクローンを検出する手法は，文・字句単位でコードクローンを検出する手法には適用可能である．提案手法の有用性が十分確認されれば，様々なコードクローン検出手法の前処理として応用が可能である．

3.2 研究目標 2「リファクタリング支援」

3.2.1 当初の想定

(1) 想定する仮説等

本テーマの研究目標は，コードクローンに対して，既存のリファクタリング支援より多くのリファクタリングパターンに対応できるツールの開発/集約方法のガイドラインの開発を行うことである． 従って，本テーマを実施するに当たって，想定する仮説は，「コードクローンの特徴に応じた集約方法で未だ対応が不十分なものが存在するかどうか」である．

(2) 当初の到達目標

上述した仮説を確認するための，到達目標としては，どのようなリファクタリングパターンを対象として支援を行うかを決めることと，対象とした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検出手法の提案とプロトタイプシステムの開発を行うことになる．

(3) 当初の期待される効果

上記の到達目標が達成されることで，評価実験とその有用性評価を実施することが可能となる．

3.2.2 研究プロセスと成果

(1) 研究プロセス

対象とするリファクタリング手法の決定，支援手法の提案，提案手法を実現したプロトタイプシステムの開発，評価実験の順に実施する．

(2) 具体的な研究成果の内容

① 対象とするリファクタリング手法

コードクローンの修正手段のひとつであるリファクタリングは，外部から見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[Fowler1999]．リファクタリングはパターンとして書籍やウェブサイト[RefWeb]にまとめられている．Fowler はメソッドの抽出，メソッドの引き上げ，フィールドの引き上げ，Template method の形成など多くのリファクタリングパターンを提案している[Fowler1999]．研究グループでも 2.1.2 で述べたようにリファクタリング支援ツールを開発しメソッドの抽出，メソッドの引き上げ等の比較的簡単なリファクタリングパターンに対する支援を行ってきた．

本テーマでは，Template Method パターンと呼ばれるデザインパターンに着目した．Template Method パターンとは，共通の親クラスを持つ類似メソッドを対象とし，メソッド間で共通の処理を親クラスに記述し,メソッドごとに異なる処理を子クラスに記述する，というパターンである．このパターンを用いたコードクローン集約手法では，メソッド間でコードクローンとなっている箇所を共通の処理として親クラスに引き上げ，コードクローンとなっていない箇所を子クラスに記述することで，コードクローンの集約を実現している．この手法の最大の特長として，対象となるメソッドがコードクローンとなっていない箇所を含んでいても適用が可能である，という点が挙げられる．この手法に基づく支援手法が幾つか提案されている[Juillerat2007][Masai2010]．しかし，既存手法には以下に挙げる課題点が存在する．

・変数名などのユーザ定義名のみが異なるコードクローンを集約できない．
・意味的に同じ処理を行っているコードでも,その表現方法が異なる場合は集約できない．

そこで本テーマではこれらの課題点を改善するため，プログラム依存グラフを用いた Template Method パターンの適用支援手法について実施する．

② 準備

1) Template Method パターン

Template Method パターンとは，Gamma らによって提案されているデザインパターンの 1 つである[Gamma1995]．このパターンは共通の親クラスを持つメソッドペアに対し，メソッド間で共通の処理を親クラスに，異なる処理をそれぞれの子クラスに記述するというものである．Template Method パターンの適用例を図 3-2-1 に示す．この例では，Site を共通の親クラスとする 2 つのクラスの間に類似メソッド getBillableAmount() が存在している．このメソッドの共通部分を親クラスに引き上げ，異なる部分をそれぞれ新たなメソッドとして抽出している．

このように修正することで，多態性によってそれぞれのクラスに応じた getBaseAmount()及び getTaxAmount() が呼び出され，適用前後で振る舞いが保たれる．このように，Template Method パターンの適用によってメソッド間に共通していた処理が親クラスにまとめて記述されるため，適用前にコードクローンとなっていたコードが集約され，ソフトウェアの保守性向上が期待できる．また，Template Method パターンはメソッド間に異なる処理が含まれていても適用可能であり，コピーアンドペースト等でコードを生成した後に修正が加えられた場合でも適用が可能であるという特長がある．

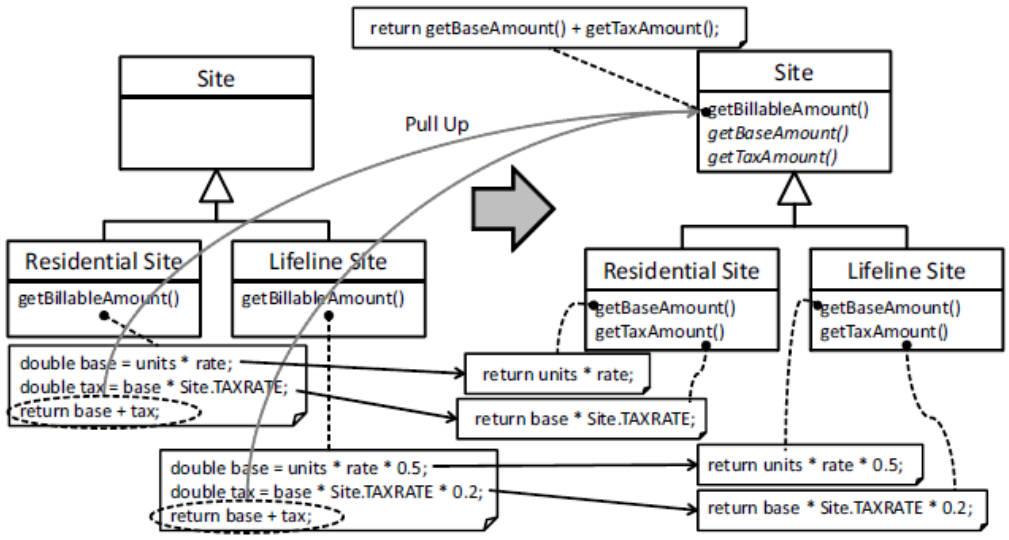


図 3-2-1　Template Method パターンの適用例

2) プログラム依存グラフ

プログラム依存グラフ(Program Dependence Graph[Ferrante1987]，以降 PDG) とは，プログラム中の要素(文) 間の依存関係を表した有向グラフである．依存関係には次の 2 種類が存在する．

データ依存：文 s で変数 v を定義し，変数 v が再定義されることなく文 t において参照される場合，文 s から文 t にデータ依存があるという．

制御依存：文 s が条件文または繰り返し文の条件式であり，その条件判定の結果によって文 t が実行されるか否かが直接決定される場合，文 s から文 t へ制御依存があるという．

図 3-2-2 の例では，4 行目で変数 y，z の値が参照されているため，2，3，5 行目から 4

行目へのデータ依存関係がある．また，5 行目は 4 行目によって実行の有無が定まるため，4 行目から 5 行目への制御依存関係がある．また，一般的に PDG はメソッドの入り口に相当する頂点を持ち，その頂点からメソッド直下のすべての文へ制御依存関係が存在する．

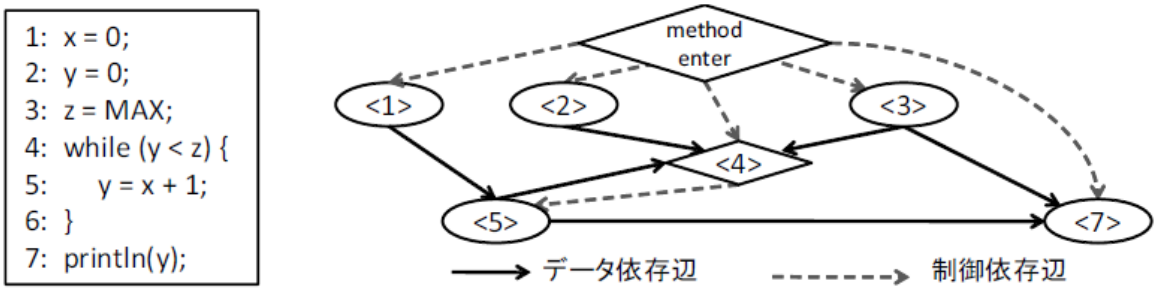


図 3-2-2　PDG の例

3）PDG を用いたコードクローン検出

これまでにコードクローンの自動検出手法は多数提案されており，それらは検出時に用いるデータ構造によって分類できる．その 1 つである PDG を用いた検出は，PDG 上で同形部分グラフとなっている箇所をコードクローンとして検出する手法である．この手法の最大の利点として，コード順序が異なる場合や，for 文と while 文等の表現の異なる場合といった，他の検出手法では検出不可能なコードクローンを検出できるという点が挙げられる．本テーマで用いる検出ツール Scorpio[Higo2010] はこの PDG を用いた検出に分類される検出ツールである．Scorpio は PDG の比較時に，各頂点が表現するコード片からハッシュ値を生成し，そのハッシュ値を用いて頂点同士の一致性を判定している．このハッシュ値の生成時に特殊な処理を行っており，ユーザ定義名のみが異なるコードクローンを検出可能である．

③　提案手法

1）概要

提案手法は，対象とするメソッドの対，各メソッドの PDG，及びそのメソッド間のコードクローン情報を入力とし，各メソッドについて親クラスに引き上げるべき頂点集合，及び子クラスに残すべき頂点集合を出力する．子クラスに残すべき頂点集合については，1 つのメソッドとして抽出すべき頂点集合も併せて出力する．

2）用語の定義

定義 3.1（対象メソッドペア）ownerclass(m) をメソッド m が実装されているクラス，ParentClasses(c) をクラス c の先祖クラスの集合であると仮定する．このとき，式(1) を満たす 2 つのメソッド(m0, m1) の対を対象メソッドペアと呼ぶ．

(ownerclass(m0) ≠ ownerclass(m1)) ∧ ∃c(c∈ ParentClasses(ownerclass(m0)) ∧ c∈ParentClasses(ownerclass(m1)))　(1)

定義 3.2（到達辺集合，出発辺集合，出発頂点，到着頂点）ある頂点 n について，n を到達頂点とする辺の集合を到達辺集合と定義し,BackwardEdges(n) と表記する．また頂点 n を出発頂点とする辺の集合を出発辺集合と定義し,ForwardEdges(n) と表記する．また，ある辺 e について，e の出発頂点を fromnode(e)，到達頂点を tonode(e) とする．

定義 3.3（PDG 頂点集合，PDG 辺集合）あるメソッド m について Nodes(m) を m の PDG 頂点集合，及び頂点集合 N について Edges(N) を N の PDG 辺集合と定義する．

定義 3.4（クローンペア）入力として与えられた対象メソッドペア(m0,m1) 間のすべてのクローンペアの集合を ClonePairs(m0,m1) とする．このとき,式(2) を満たす頂点集合 N0, N1 の対をクローンペアと呼び，cp(N0,N1) と表記する．

(N0,N1) ∈ ClonePairs(m0,m1) (2)

定義 3.5（共通頂点集合，メソッド内共通頂点集合）対象メソッドペアの共通頂点集合 CommonNodePairs(m0,m1) を式(3) で定義する．

CommonNodePairs(m0,m1) := {(s,t) : ∃cp(N0,N1) ∈ ClonePairs(m0,m1)((s ∈ N0) ∧ (t ∈ N1))} (3)

また，対象メソッドペア(m0, m1) の各メソッドについて，そのメソッドのメソッド内共通頂点集合 CommonNodes(mi) (i = 0, 1) を式(4) で定義する．

CommonNodes(mi) := {n 2 Nodes(mi) : (∃(s,t) ∈ CommonNodePairs (m0,m1) ((s = n)∨(t = n)))} (4)

定義 3.6（差異頂点集合）対象メソッドペア(m0, m1) の各メソッドについて，そのメソッドの差異頂点集合 DiffNodes(mi) (i = 0, 1) を式(5) で定義する．

DiffNodes(mi) := {n ∈ Nodes(mi) : n ∉ CommonNodes(mi)} (5)

3) 処理

ここでは提案手法の処理内容について述べる．なお，提案手法は下記の手順で実行される．

Step1 Template Method パターンの適用に必要な条件の解決．

Step2 1 つのメソッドとして抽出すべき頂点集合の特定．

Step3 Step2 で特定した頂点集合同士のメソッドペア間での対応付け．

以降，各ステップについて詳細に述べる．

Step1:必要条件の解決

Template Method の形成にはいくつかの必要条件が存在するため，対象メソッドペアへの適用が可能か否かを判定する必要がある．適用ができない場合，共通頂点集合を必要条件を満たすように操作し，共通頂点集合の操作では必要条件の解決ができない場合は適用が不可能であると判断する．以下に満たさなければならない必要条件とその解決方法を述

べる.

条件 1: 差異頂点集合が return 文を含まない

差異頂点集合が表現する文に return 文が含まれる場合，差異頂点集合を新規メソッドとして抽出した際，作成したメソッドはそのメソッド自身とそのメソッドを呼び出す親クラスに引き上げたメソッドの 2 つのメソッドを同時に脱け出さなければならない．このような場合，提案手法は入力されたメソッドペアに対しての Template Method パターン適用は不可能と判定する．すなわち，statement(n) を頂点 n が表現する文，return(s) を文 s が return 文のときのみ真となる述語論理式であるとするとき，以下の論理式 hasReturn(m0,m1) を充足する対象メソッドペア(m0, m1) を適用不可能であると判定する．

hasReturn(m0,m1):=∃i∈{0,1}(∃n∈DiffNodes(mi)(return(statement(n)))) (6)

条件 2:差異頂点集合からメソッド内共通頂点集合への制御依存が存在しない

control (e) を辺 e が制御依存辺のときのみ真となる述語論理式であるとする．このとき，対象メソッドペア(m0, m1) について，論理式 invalidControl (m0, m1) を以下に定義する．

invalidControl(m0,m1) := ∃i∈{0,1} (∃e ∈Edges(Nodes(mi))((control(e)) ∧ (fromnode(e) ∈ DiffNodes(mi)) ∧ (tonode(e) ∈ CommonNodes(mi)))) (7)

invalidControl (m0, m1) を充足する対象メソッドペア(m0, m1) の差異頂点集合を新規メソッドとして抽出することは不可能である．これは図 3-2-3 のように，差異頂点集合が条件式とブロック文の一部のみを含んでいる場合に生じる問題である．この条件は共通頂点集合に含む頂点ペア情報を操作することで解決が可能である．図 3-2-4 にその解決方法 resolveIC (CommonNodePairs(m0, m1)) を示す．

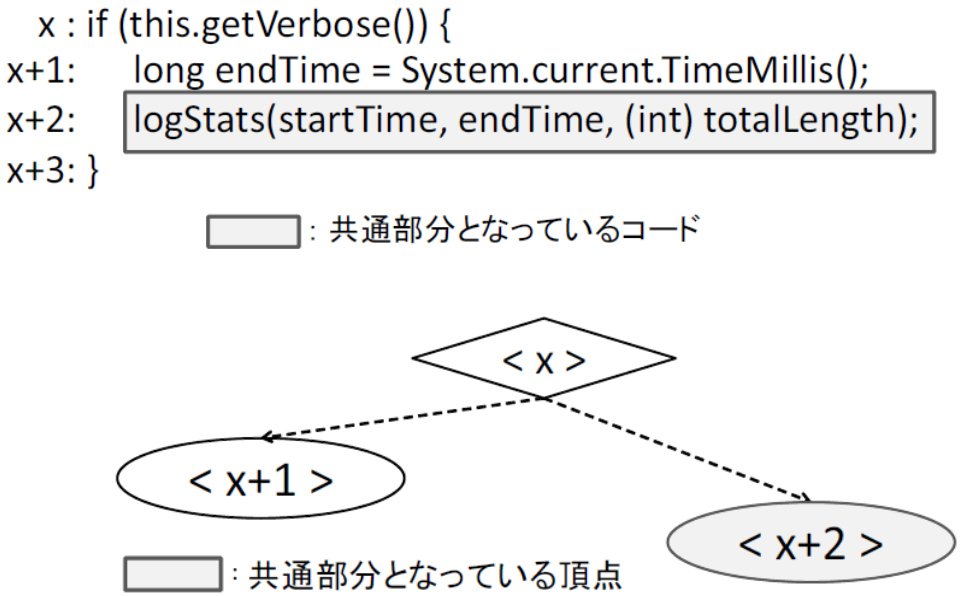


図 3-2-3 差異部分から共通部分への制御依存

```
Input: CommonNodePairs(m0, m1 )
Output: CommonNodePairs(m0, m1 )(¬invalidControl(m0,m1))
for all (n0,n1) such that (n0, n1) ∈ CommonNodePairs(m0, m1 ) do
  for all be such that ((be ∈ BackwardEdges(n0)) ∧control(be) ∧fromnode(be)
    ∈ DiffNodes(m0))
  do
    CommonNodePairs(m0,m1 )← CommonNodePairs(m0,m1 )- (n0, n1)
  end for
  for all be such that ((be ∈ BackwardEdges(n1)) ∧control(be) ∧fromnode(be)
    ∈ DiffNodes(m1))
  do
     CommonNodePairs(m0,m1 ) ← CommonNodePairs(m0,m1 )-(n0, n1)
  end for
end for
return CommonNodePairs(m0, m1 )
```

図 3-2-4 resolveIC (CommonNodePairs(m0, m1))

Step2:抽出頂点集合，抽出頂点集合群の特定

対象メソッドペア(m0, m1) の各メソッドについて，その差異頂点集合から 1 つの新規メソッドとして抽出すべき頂点の集合である抽出頂点集合を特定し，それらの集合である抽出頂点集合群 DiffGroups(mi) (i = 0; 1) を特定する．まず，頂点集合 N に含まれる頂点のうち，頂点 n から辺を辿って到達可能な頂点集合を
特定する処理 search(n, N) を以下に定義する．

```
Input: n:基点, N:探索可能頂点集合
Output: n から辺を辿って到達可能かつ N に含まれる頂点集合
T ← n
for all e such that e ∈ ForwardEdges(n) do
   if tonode(e) ∈  N then
     T ←  T + search(tonode(e), N)
   end if
end for
return T
```

次に，2 つの頂点集合 S; T に対して merge(S, T) を以下に定義する．

$$merge(S, T) := S \cup T \tag{8}$$

このとき，DiffGroups(mi) を特定する処理 detectDiffGroups(mi) を以下に示す．

```
Input: DiffNodes(mi)
Output: DiffGroups(mi)
DiffGroups(mi) ← ∅ ;
for all n such that n ∈ DiffNodes(mi) do
   dg ←   search(n, DiffNodes(mi))
   DiffGroups(mi) ←  dg
   for all adg such that (adg ∈ DiffGroups(mi) ∧ ∃n ∈dg(n ∈ adg)) do
        DiffGroups(mi) ← DiffGroups(mi) - adg + merge(adg, dg)
   end for
end for
```

Step3:抽出頂点集合同士の対応付け

対象メソッドペア(m0, m1) の各メソッドにおける抽出頂点集合群 DiffGroups(m0), DiffGroups(m1) の各要素同士で対応を取り，抽出頂点集合ペア DiffGroupPairs(m0;m1)を特定する．まずこの手順に必要となる返り値集合の特定手法を述べた後，この手順のアルゴリズムを説明する．

・返り値集合の特定

ある抽出頂点集合 dg をメソッドとして抽出した際，返すべき値の集合である返り値集合 ReturnValues(dg) を特定する．はじめに，data(e) を辺 e がデータ依存辺のときのみ真となる論理式とするとき,頂点集合 N からのデータ依存辺の集合 output(N) を式(9) で定義する．

output(N):= {e∈Edges(N): ((fromnode(e)∈N)∧(tonode(e) ∉ N)∧ data(e))} (9)

また，頂点集合 N を保持するメソッドを ownermethod(N)，Variables(m) をメソッド m で使用される変数の集合とし，var(de) をデータ依存辺 de が表す変数とする．このとき，返り値集合 ReturnValues(dg) を式(10) で定義する．

ReturnValues(dg) := {v∈Variables(ownermethod(dg)): ∃e ∈output(dg)(v = var(e))} (10)

・対応付けアルゴリズム

以下の説明では，対象メソッドペアを(m0, m1) とし，各メソッドを mi (i = 0, 1) とする．

はじめに，ある抽出頂点集合 N に対し，その抽出頂点集合へのデータ依存辺を持ち，かつそのデータ依存辺の出発頂点がメソッド内共通頂点集合に含まれるような辺集合 input(N, mi)を式(11) で定義する．

input(N,mi) := {e ∈ Edges(Nodes(mi)) : ((fromnode(e) ∈ CommonNodes(mi)) ∧ (tonode(e) ∈ N))} (11)

また，2 つの頂点 n0, n1 の対が共通頂点集合に含まれるか否かを判定する式 clone(n0, n1)を式(12) で定義する．

clone(n0, n1) := ∃(a, b) ∈ CommonNodePairs (m0,m1) ((n0 = a∧ n1 = b) ∨(n0 = b∧ n1 = a)) (12)

次に，typeE(e) を辺 e の種類(データ依存辺か制御依存辺か)，typeV (v) を変数 v の型を表すものとし，|S|を集合 S の要素数とする．このとき，2 つの抽出頂点集合 DG0,DG1 への依存辺の集合 B0, B1 に対し，それらの依存辺の出発頂点が共通頂点集合内で対応しているか否かを判定する論理式 sameInput(B0,B1) を式(13) で定義する．また,DG0,DG1 からのデータ依存辺が表す変数の集合 V0, V1 に対し，それらの変数が表す型及び変数の数が一致するか否かを判定する式 sameOutput(V0,V1) を式(14) で定義する．

sameInput(B0,B1) := (∀b0 ∈ B0, ∃ b1 ∈ B1((typeE(b0) = typeE(b1)) ∧ clone(fromnode(b0), fromnode(b1)))(∀b1 ∈ B1, ∃ b0 ∈ B0((typeE(b0) = typeE(b1)) ∧clone(fromnode(b0),fromnode(b1))) (13)

sameOutput(V0,V1) := (|V0| = |V1|)∧(∀v0 ∈ V0, ∃v1(V1(typeV (v0) = typeV (v1)))∧(∀v1 ∈ V1; ∃v0 ∈ V0(typeV (v0) = typeV (v1))) (14)

このとき,抽出頂点集合間の対応の有無を判定する式 same(S, T) を式(15) に定義する．

same(S, T) := sameInput(input(S, ownermethod(S)), input(T, ownermethod(T))) ∧ sameOutput(ReturnValues(S), ReturnValues(T)) (15)

以上の定義のもとで，抽出頂点集合の対応を取る処理 detectDGPairs(DiffGroups(m0), DiffGroups(m1)) を以下に示す。

```
Input: DiffGroups(m0); DiffGroups(m1)
Output: DiffGroupPairs(m0;m1)
DiffGroupPairs(m0,m1)←⌀
RDG←⌀
for all dg0 such that dg0 ∈ DiffGroups(m0) do
    pd←false
    for all dg1 such that dg1 ∈ DiffGroups(m1) do
        if same(dg0, dg1) then
            DiffGroupPairs(m0,m1)←DiffGroupPairs(m0,m1) + (dg0, dg1)
            RDG←RDG + dg1
            Pd←true
            break
        end if
    end for
    if pd = false then
```

```
            DiffGroupPairs(m0,m1)←DiffGroupPairs(m0,m1) + (dg0, NULL)
        end if
    end for
    for all dg1 such that dg1 ∈ DiffGroups(m1) ∧ dg1 ∉ RDG do
         DiffGroupPairs(m0,m1)←DiffGroupPairs(m0,m1) + (NULL, dg1)
    end for
    return DiffGroupPairs(m0,m1)
```

ここで対応関係の取られた抽出頂点集合 dgpair(dg0, dg1) ∈ DiffGroupPairs(m0,m1) は同じシグネチャを持つメソッドとして抽出すべき箇所となる．いずれか一方が NULL の場合，その処理は一方のメソッドにしか含まれない処理であり，その処理を含まないメソッドには同じシグネチャを持つ何もしないメソッドを作成する必要が生じる．

4) 処理の流れ

提案手法の処理の流れを以下に示す．なお，入力情報から CommonNodePairs(m0,m1) を特定する処理を detectInitCN とし，CommonNodePairs(m0,m1) から DiffNodes(mi) (i =0, 1) を特定する処理を detectDN と表記している．

```
Input:(m0,m1),Nodes(m0),Nodes(m1),Edges(Nodes(m0)),Edges(Nodes(m1)),
       ClonePairs(m0,m1)
Output:(CommonNodePairs(m0,m1),DiffGroupPairs(m0,m1))
DiffGroupPairs(m0;m1) ←∅
CommonNodePairs(m0,m1)←detectInitCN
DiffNodes(m0)←detectDN(m0)
DiffNodes(m1)←detectDN(m1)
if hasReturn(m0,m1) then
   return (NULL, NULL) /* invalid method pair */
end if
if invalidControl(m0,m1) then
   CommonNodePairs(m0,m1)←resolveIC (CommonNodePairs(m0,m1))
   DiffNodes(m0)←detectDN(m0)
   DiffNodes(m1)←detectDN(m1)
end if
DiffGroups0←detectDiffGroups(m0)
DiffGroups1←detectDiffGroups(m1)
DiffGroupPairs(m0,m1)←detectDGPairs(DiffGroups0,DiffGroups1)
return CommonNodePairs(m0,m1), DiffGroupPairs(m0,m1)
```

④ 実装

提案手法を Java を用いて実装した．入力となる PDG 情報及びコードクローン情報の特定には，ソースコード解析ツール MASU[Miyake2009]) とコードクローン検出ツール

Scorpio を用いた.

1)入力支援

提案手法では入力として対象となるメソッドペアをあらかじめ指定する必要がある．しかし，膨大なソースコードから対象メソッドペアを特定することは困難であると考えられる．このため，本実装では Scorpio で検出されたコードクローン情報から適用可能な候補を自動的に特定し,その一覧を利用者に提示することで適用可能候補の特定の支援を行う.

2)言語依存支援

提案手法はプログラミング言語に依存せずに適用可能であるが，適用対象のプログラミング言語に応じた処理を追加することでより的確な支援が実現可能であると考えられる．MASU 及び Scorpio が Java のみに対応しているため,本実装ではパターン適用時の必要条件に Java に特化した条件を追加した．以降で追加した条件について述べる.

条件 3:抽出頂点集合が返すべき値が 1 つ以下である

Java では 1 つのメソッドは 1 つ以下の値しか返すことができない．このため，提案手法で特定した抽出頂点集合が新規メソッドとして抽出不可能である可能性がある．そこで本実装では，③3) で述べた抽出頂点集合特定処理の終了後に処理を追加する.

まず，ある頂点集合 N について，Bounds(N) を式(16) で定義する.

$$Bounds(N) := \{n \in N : (\exists e \in ForwardEdges(n)(tonode(e) \notin N))\} \quad (16)$$

次に，NodesContainEdgesOfOther(N, v) を式(17) で定義する.

$$NodesContainEdgesOfOther(N, v) := \{n \in N : \exists e \in ForwardEdges(n)(data(e) \wedge (tonode(e) \notin N) \wedge (var(e) \neq v))\} \quad (17)$$

このとき，次に示す処理を DiffGroups(mi) に施すことで条件の解決を行う.

```
Input: DiffGroups(mi)
Output: DiffGroups(mi)(sartisfying∀dg∈DiffGroups(mi)(|ReturnValues(dg)|<=1))
for all dg such that dg ∈ DiffGroups(mi) do
  if (jReturnValues(dg)j > 1) then
    max ←0
    U ←∅;
    for all v such that v ∈ ReturnValues(dg) do
      S ←dg
      T ←∅
      repeat
          T←Bounds(S)
          S←S - NodesContainEdgesOfOther(T, v)
      until (|NodesContainEdgesOfOther(T, v)| = 0)
      if |S|≧  max then
```

```
            max = |S|
            U←S
        end if
    end for
    DiffGroups(mi)←DiffGroups(mi) - dg + U
    RN ←dg - U
    for all n such that n ∈ RN do
        tdg ←search(n,RN)
        for all atdg such that (atdg∈ DiffGroups(mi) ∧ ∃a ∈ tdg(a ∈atdg)) do
            DiffGroups(mi)←DiffGroups(mi)-atdg + merge(atdg, tdg)
        end for
     end for
   end if
end for
```

条件 4:抽出頂点集合がブロック文をまたぎ，かつその一部分のみしか含んでいない

この条件は条件 2 と類似する条件だが，こちらは try 文などの，条件式を含まないブロック文によって生じる問題である．条件式を含まないブロック文の場合は制御依存関係が発生しないため，PDG の情報のみを用いてこの条件を満たすか否かを判別することはできない．そこで本実装では，MASU から得た構文情報を用いて条件の解決を行う．まず,③3) において PDG を走査する search(n,N) を実行する際に,sameBlock(f, t) という条件を追加する．すなわち，search(n,N) を以下のように変更する．

```
Input: n:基点, N:探索可能頂点集合
Output: n から辺を辿って到達可能かつ N に含まれる頂点集合
T←n
for all e such that e ∈ ForwardEdges(n) do
    if tonode(e) ∈ N ∧ sameBlock(fromnode(e),tonode(e)) then
       T ←  T + search(tonode(e),N)
    end if
end for
return T
```

ここで，頂点 n を直接保持するブロックを ownerblock(n) とするとき，2 つの頂点 f, t を保持するブロック文の一致性を判定する式 sameBlock(f; t) を式(18) に定義する．

$$sameBlock(f,t) := (ownerblock(f) = ownerblock(t)) \quad (18)$$

この制約を追加することで，同じブロック内に含まれる頂点のみが抽出頂点集合として特定されるため，前述の問題点を解決することが可能となる．しかし，この制約を追加することで，抽出頂点集合の大きさが各ブロック文までに制限されてしまい，抽出頂点集合の数が増大してしまうという課題点が生じる．図 3-2-5 の場合，制約の追加前はすべての文が1つの抽出頂点集合に含まれるが,制約を追加することでこれらが2つに分断される．

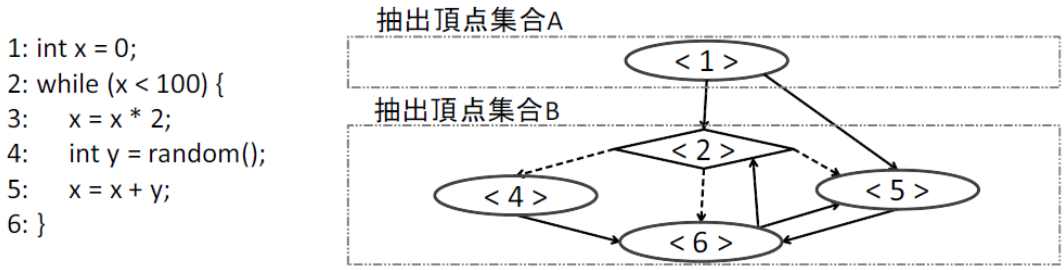


図 3-2-5　制約の追加による抽出頂点集合の分離問題

この問題を解決するために，ブロック文の集約処理をさらに追加する．以降このブロック文集約処理について述べる．あるブロック文 s について，そのブロックを構成する頂点集合を BlockNodes(s) とする．このとき，ブロック文の全頂点が最頂点集合に含まれるかを判定する式 allDN(s,mi) を式(19) に定義する．

$$allDN(s) := \forall n \in BlockNodes(s)(n \in DiffNodes(mi)) \quad (19)$$

また，BlockStatements(m) をメソッド m のブロック文の集合とする．以上の定義を用いて，③3) の処理を行う直前に以下に述べる処理を追加する．

```
for all s such that BlockStatements(mi) do
  if allDN(s,mi) then
      n←mergeNode(BlockNodes(s))
      DiffNodes(mi) ←DiffNodes(mi) - BlockNodes(s) + n
  end if
 end for
```

ここで，owner(N) を頂点集合 N を直接保持するブロック文とするとき，mergeNode(N) は頂点集合 N を基に以下の条件を満たす単一頂点 n を生成する処理である．

$$BackwardEdges(n) = \{e \in Edges(N) : ((fromnode(e) \in N) \wedge (tonode(e) \notin N))\} \quad (20)$$

$$ForwardEdges(n) = \{e \in Edges(N) : ((fromnode(e) \notin N) \wedge (tonode(e) \in N))\} \quad (21)$$

$$ownerblock(n) = owner(BlockNodes(owner(N))) \quad (22)$$

3)候補の提示

実装したツールは適用対象の PDG とソースコードの提示機能を有している．ソースコード表示では，共通部分をハイライト表示し，差異部分については 1 つのメソッド内に抽出すべき文を同じ色枠で囲んで表示することで集約の支援を行う．また，対象メソッドペアにメソッド間の類似度などのメトリクスをいくつか定義し，利用者がそれらのメトリクスの閾値を自由に設定できる機能を実装した．これは，利用者によって集約したいと考える条件が異なると考えられるため，利用者が自由に候補を選定可能にすることで，集約したいと思える候補がより発見しやすくなると考えたためである．

⑤ コード評価

開発したプロトタイプシステムを，3 つのオープンソースソフトウェアと IT Spiral 実プロジェクトデータに対して適用した．適用結果を表 3-2-1 に示す．

表 3-2-1　適用結果

Target Systems	LOC	# of Candidates	Elapsed Time[s]
Apache Ant	212,401	226	237
Argo UML	328,582	486	1,080
Apache Synapse	58,418	45	95
IT Spiral	31,948	9	45

規模 3 万行～32 万行程度のソフトウェアに対して，表 3-2-1 に示す時間で，Template Method パターンが適用可能なコードクローンを検出できた．図 3-2-6 に抽出した例を示す．図 3-2-7 に，IT Spiral 実プロジェクトデータから検出結果のプロトタイプシステム画面を示す．また，Apache Synapse から検出した 45 個の候補に対して，Template Method パターンを適用して集約を行った．1 個あたり 10 分程度で集約を行え，集約前後で動作が変わらないことを確認した．本手法を用いて提示されたコードクローンに対する Template Method パターンの適用手順は，リファクタリングのガイドラインとして十分利用可能であると判断できる．

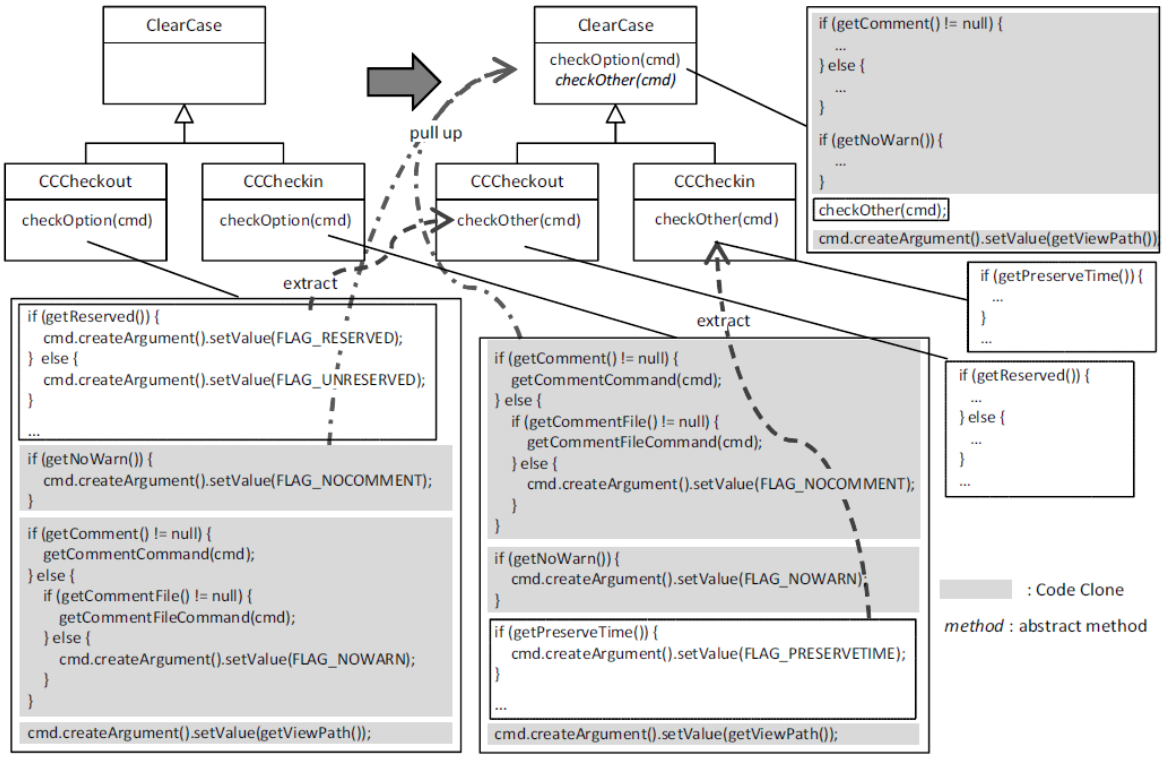


図 3-2-6　適用例

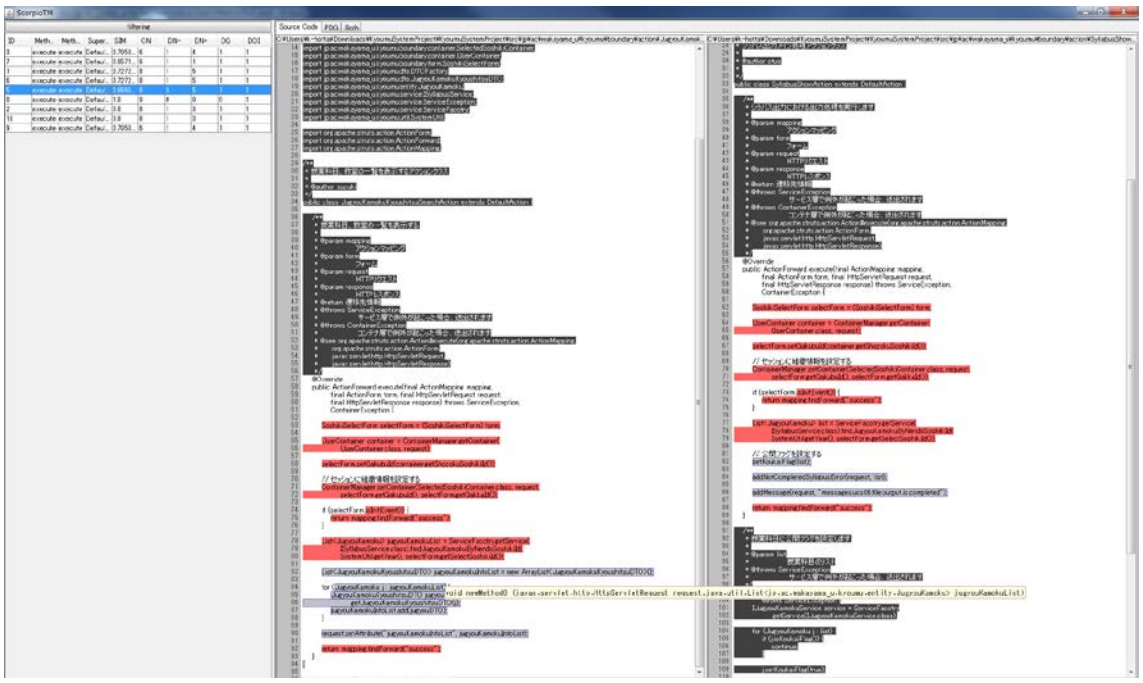


図 3-2-7　適用結果画面例

3.2.3 実用化へ向けた課題と問題点

(1) 課題と問題点

本テーマでは，Template Method パターンと呼ばれるデザインパターンが適用可能なコードクローンを，プログラム依存グラフを用いて検出する手法を提案した．提案手法を用いたコードクローン検出ツールを実装し，オープンソースソフトウェアに対して検出を行った．その結果，Template Method パターンを適用可能なコードクローンを検出でき，一部については実際にリファクタリングを実施できた．

今後の課題として，パターン適用後のソースコード例の提示，及び3つ以上の類似メソッドへの拡張などが挙げられる．

(2) 将来の応用方法

提案手法は，完成後のソフトウェアに対して適用を行ったが，リファクタリングは開発途中で積極的に行うべきものであるという指摘もある．本手法や既存のコードクローン検出ツールを用いることで，開発途中でのリファクタリング適用を効果的に行うことが可能であると考える．開発現場での実際の利用に期待したい．

3.3 研究目標3「再利用ライブラリ作成支援」,「違反流用コード発見支援」

テーマ3とテーマ4は，実現手段や評価実験が共通して実施できたため，一つにまとめて報告を行う．

3.3.1 当初の想定

(1) 想定する仮説等

本テーマの研究目標は，大規模なソースコード群から，適切な大きさ（例えば，メソッド，関数単位）のコードクローンを検出することで，再利用ライブラリに有用なものや違反流用コードを抽出することである．

本テーマを実施するに当たって，想定する仮説は,「適切な大きさのコードクローン検出することで，再利用ライブラリに有用なものや違反流用コードを抽出できるか」ということになる．

(2) 当初の到達目標

上述した仮説を確認するための，到達目標としては，特定のプログラミング言語で実装されたソースコード群を対象として，適切な大きさのコードクローンを検出するプロトタイプシステムを開発することになる．

(3) 当初の期待される効果

上記の到達目標が達成されることで，評価実験とその有用性評価を実施することが可能となる.また,再利用可能なコードクローンや違反流用コードを効率よく検出可能となる．

3.3.2 研究プロセスと成果

(1) 研究プロセス

コードクローンの大きさの設定，設定した大きさのコードクローンの検出手法の提案，提案手法を実現したプロトタイプシステムの開発，評価実験の順に実施する．

(2) 具体的な研究成果の内容

① コードクローンの大きさの設定

複数のソフトウェアにまたがるコードクローンを検出することは，複数のソフトウェア間に頻出する処理のライブラリ化による開発効率の向上やライセンスに違反して流用されているソースコードの特定などの観点から有益である．しかし，既存のコードクローン検出手法は１つのソフトウェア内のコードクローンを見つけることを目的としているため，ソースコードを文や字句などの細粒度で比較している．そのため，大規模なソフトウェア群を対象としたコードクローン検出には膨大な時間的・空間的コストを必要とする．この問題を改善し，大規模なソフトウェア群から実用的なコストでコードクローンを検出する手法として，ファイル単位のコードクローン検出手法が提案されている[Sasaki2011][Ossher2011]．ファイル単位のコードクローン検出手法は，大規模なソフトウェア群に対して高速にコードクローン検出を行うことができる反面，ファイルの一部がコードクローンであるものは検出できないという問題点を抱えている．例えば，ファイルのある一部分のみが流用されている場合，ファイル単位のコードクローン検出手法では流用部分をコードクローンとして検出することができない．そこで，本テーマでは，ファイル単位より小さい粒度であるメソッド単位でのコードクローン検出を行う．メソッド単位でのコードクローン検出手法では，大規模なソフトウェア群に対して実用的な時間で検出を行うことができると同時に，ファイル単位のコードクローン検出手法では検出できない，ファイルの一部がコードクローンであるものを検出することが可能であることが期待できる．

② メソッド単位のコードクローン検出手法

ここでは，大規模なソフトウェア群からメソッドクローンを検出する方法について述べる．

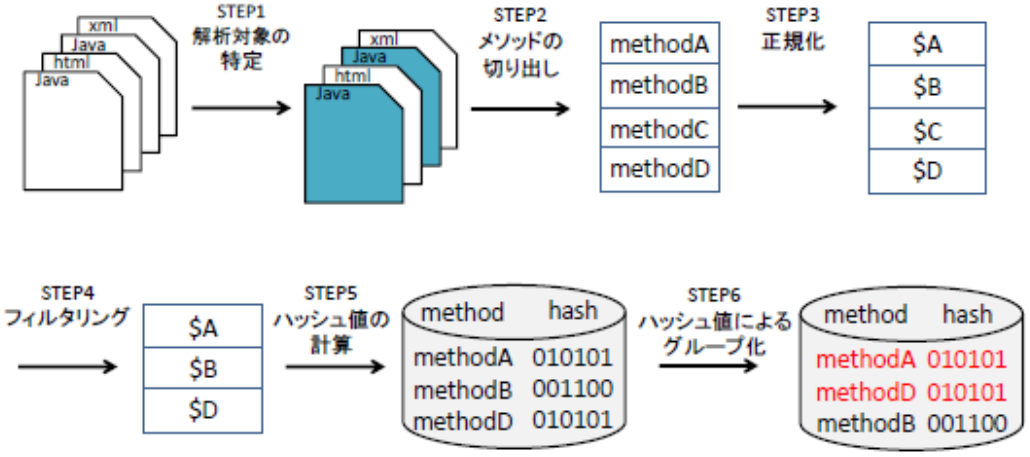


図 3-3-1　検出手法概要

対象とするデータセットからその中に存在するメソッドクローンとファイルクローンを検出する．図 3-3-1 に検出処理の概要を示す．以降，図 3-3-1 に示す各処理について詳細に述べる．

STEP1：解析対象の特定

入力として与えられたデータセットから，解析対象となるソースファイルを特定する．今回の調査では，Java を用いて記述されたソフトウェアのみを解析対象としているため，入力として与えられたデータセットから拡張子が.java であるファイルを特定する．

STEP2：メソッドの切り出し

STEP1 で得られたソースファイルから，メソッドを切り出す．今回の調査では，ソースファイルから抽象構文木を作成してメソッドの切り出しを行う．

STEP3：正規化

このステップでは，コメント文の有無や空白・改行回数などの違いを取り除くために正規化を行う．具体的には，以下に示す正規化処理を行う．

「変数名，文字列リテラル，メソッド・クラス・インターフェース宣言部のメソッド名は 1 つの特殊な文字列に置換する」

「空白，改行，修飾子，アノテーション，コメント文，インポート文，パッケージ文は削除する」

STEP4：フィルタリング

ゲッタメソッドやセッタメソッドのように，処理が単純でありかつ短いメソッドは多くのソースファイルに存在すると考えられる．そのため，このようなメソッドは大量にコー

ドクローンとして検出されるおそれがある．このようなコードクローンは，流用の特定や複数のソフトウェアに存在する共通処理のライブラリ化に対して有用ではない．したがって，処理の単純かつ短いメソッドは検出が不要である．このステップでは，このようなメソッドを検出の対象から除外する．今回の調査では，メソッド内の複文の数が 0 であるメソッドについては検出の対象から除外する．ここで複文とは複数の文から成り立つブロックを指し，do，for，if，switch，try，while 文を複文と定義する．

STEP5：ハッシュ値の計算

解析対象となるメソッドそれぞれに対して，ハッシュ値を算出する．今回の調査では，MD5[MD5] によりハッシュ値を計算する.等しい記述を持つメソッドはハッシュ値が等しくなるため，ハッシュ値が等しいメソッドはコードクローンと考えられる．算出されたハッシュ値は，メソッドごとデータベースに格納する．

STEP6：ハッシュ値によるグループ化

ハッシュ値の等しいメソッドをグループ化する．それらのメソッドグループのうち，要素数が 2 以上のグループがメソッドクローンとして検出される．

③ 実験 1

1) 実験対象

本研究では，実験対象として"UCI source code data sets" [Ossher2011] [Lopes](以降，UCIdatasets と呼ぶ) を用いた．しかし，UCIdatasets には trunk,tags,branches を同時に含むソフトウェアやバージョンが違う同一ソフトウェアが複数含まれている．ソフトウェアは処理の追加，修正，削除を行い，新しいバージョンに進化していく．しかし，処理の修正などが行われないソースコードは，バージョンが新しくなってもその内容が変更されないため，コードクローンとして検出される．したがって，このようなソフトウェアに対してコードクローン検出を行うと，検出されるコードクローンが不必要に多くなるおそれがある．そこで本研究では，trunk,tags,branches を同時に含むソフトウェアは trunk 以下のファイルのみを，バージョンの違う同一ソフトウェアは最新バージョンのみを検出対象とした．また，いくつかのソフトウェアにはソースコード自動生成ツールによって作成されたソースコードが存在した．このようなソースコードはコードクローンとして検出されるが，検出されたコードクローンは流用の特定や複数のソフトウェアに存在する共通処理のライブラリ化に対して有用ではない．そのため本研究では，ソースコード自動生成ツールによって作成されたソースコードは検出の対象から除外した．さらに，UCIdatasets から今回解析対象とする拡張子が.java であるファイル以外を削除している．このような処理を行うことで，解析対象以外のファイルに対して解析対象であるかを判定する必要がなくなり，検出速度が高速化する．

表 3-3-1 は，UCIdatasets の構成を表したものである．上記処理の結果，検出対象ファイル数が全ファイル数の約半数に減少している．

表 3-3-1　適用対象

全ファイル数	3,963,896
検出対象ファイル数	2,072,490
ソフトウェア数	13,193
検出対象メソッド数	5,953,165
検出対象ファイルの総行数	361,663,992
全容量	30.6GByte

2）適用結果

UCIdatasets に対して，実装したツールを適用してコードクローンを検出した．メソッドクローン数は 2,937,047，メソッドクローンの種類数は 814,391 となった．814,391 種類のうち，約 60%にあたる 490,206 が複数のソフトウェアにまたがっていた．表 3-3-2，表 3-3-3 は，それぞれメソッドクローンとコードクローンを含むファイルの分析結果である．また表中の百分率は，それぞれ検出対象メソッド数，検出対象ファイル数に対する割合になっている．表 3-3-2 より，ファイルクローンでないファイルに存在するメソッドクローンは，検出された全メソッドクローン 2,937,047 の 40%を占めることがわかる．また表 3-3-3 より，ファイルクローンではないがメソッドクローンを含むファイルは，コードクローンを含む全ファイル 1,079,789 の 27%を占めることがわかる．

表 3-3-2　メソッドクローン数

	ソフトウェアにまたがる	ソフトウェアにまたがらない	合計
ファイルクローンの一部	1,407,338(24%)	365,150(6%)	1,772,448(30%)
ファイルクローンでない	658,500(11%)	506,059(9%)	1,164,559(20%)
合計	2,065,838(35%)	871,209(15%)	2,937,047(49%)

表 3-3-3　コードクローンを含むファイル数

		ソフトウェアにまたがる	ソフトウェアにまたがらない	合計
ファイルクローン	検出対象メソッドを含む	288,185(14%)	84,213(4%)	372,398(18%)
	検出対象メソッドを含まない	304,779(15%)	114,412(6%)	419,191(20%)
	合計	592,964(29%)	198,625(10%)	791,589(38%)
ファイルクローンではないがメソッドクローンを含む		147,532(7%)	140,668(7%)	288,200(14%)
合計		740,496(36%)	339,293(16%)	1,079,789(52%)

3）発生原因

検出したメソッドクローンにどのようなものが存在するかを分析するために，またがっているソフトウェア数上位 100 種類のメソッドクローンを調査した．調査の結果を表 3-3-4 に示す．表 3-3-4 より，上位 100 種類のメソッドクローンのうち 40%が GUI 関連の処理を行うメソッドであることがわかる．また，GUI 関連の処理をするメソッドのうち約半数は，SwingWorker などの抽象クラス内もしくは AbstractTableModel などの抽象クラスを継承したクラス内で宣言されているメソッドであることがわかった．

表 3-3-4　メソッドクローン分類

メソッドの種類		検出数
GUI 関連	AbstractTableModel	15
	その他 Table 関連	10
	SwingWorker	7
	その他	8
64bit-encorder,decorder		13
FileFilter		7
ResourceBundle		4
その他		36

AbstractTableModel は Table 関連の処理を行うクラスであり，SwingWorker は GUI における時間のかかる処理を別のスレッドで実行させるためのクラスである．また，他のソフトウェアとコードクローンを共有しているファイルにどのようなものがあるかを分析するため，コードクローンを共有しているソフトウェア数上位 500 ファイルを調査した．調査の結果，約 330 ファイルが SwingWorker クラス，約 150 ファイルが ResourceBundle を使用するクラス，残り 20 ファイルほどが GUI 関連のメソッドクローンを内包しているファイルであった.ResourceBundle はプロパティファイルに記述されているデータを読み込むために使用される．

実験の結果から，メソッドクローンの発生原因として以下の要因が挙げられる．

・抽象クラスの継承，インターフェースの実装

抽象クラスを継承するクラスやインターフェースを実装するクラスでは，全ての抽象メソッドをオーバーライドしなければならない．またこのようなクラスでは，新たに処理を記述する際に別々のソフトウェアであっても処理が類似することがある.その結果，多くのソフトウェアにまたがるコードクローンが発生すると考えられる．

・ソースコードの流用

ソフトウェア開発では，web 上などで公開されているソースコードを開発中のソフトウェアに使用することがある．このとき，自分の作成したソースコードに対応させるといった理由で，使用するソースコードに変更を加えることが考えられる．その結果，ソ

ースコードの一部のメソッドがコードクローンとして検出される.

・汎用的な処理を行うメソッド

size やclose といったメソッドは多くのクラスで定義されている. このようなメソッドは,特定の変数が null や0 であるといった条件判定と併用されやすい.そのため, 上述したフィルタリング処理を通過しコードクローンとして検出される.

4) 解析速度

FCFinder [Sasaki2011]では, 1CPU,1core(2.50GHz), メモリ 4GByte の環境で, 11.2GByte のソースコードに対して 17.16 時間でコードクローンの検出を終えている. 一方今回の実験では, 1CPU,4core(2.00GHz), メモリ 8GByte の環境で実装したツールを UCIdatasets に適用した結果, 4.91 時間でコードクローンの検出が完了した. 表 3-3-5 に各処理に要した時間を示す. このような検出時間でコードクローン検出が完了した理由として, データベースを SSD 上においているためデータベースアクセス速度が高速であることや, UCIdatasets から今回解析対象とする拡張子が.java であるファイル以外を削除していることが挙げられる. 実行環境やソースコードのサイズが異なるため厳密な比較は困難であるが, 十分に実用的な時間で検出が完了したものと考えられる.

表 3-3-5 解析時間

処理内容	時間
対象ファイルの特定	0.46h
メソッドの切り出し 正規化 フィルタリング ハッシュ値の計算	2.98h
ハッシュ値のグループ化	1.47h
合計	4.91h

5) 考察

検出したメソッドクローンが有用であるかを評価するために, ファイルクローンとしては検出されず, かつ流用の特定や処理のライブラリ化が実現できそうなメソッドクローンが存在するかを調査した. 流用の特定はまたがっているソフトウェア数の下位, 処理のライブラリ化はまたがっているソフトウェア数の上位のメソッドクローンを調査した. 図 3-3-2, 図 3-3-3 は実装したツールが検出したメソッドクローンの例である. まず図 3-3-2 について, ハイライトされている部分が検出されたメソッドクローンである. 図 3-3-2 (a) のソースコードはファイルの先頭にライセンスについての記述があったが, 図 3-3-2 (b) のソースコードは記述が存在しなかった. また, 図 3-3-2 (a) のソースコードでは stringConverter という変数を宣言し, put メソッドで stringConverter を呼び出してい

る．一方，図 3-3-2 (b) のソースコードでは put メソッドの内部で処理を記述している．いずれのソースコードも処理の内容自体は変わらないため，違いは記述方法のみである．以上二点から，図 3-3-2 はソースコード流用の一例であると考えられる．次に図 3-3-3 について，2 つのソースコードは Table に関する処理を行うクラスで宣言されているソートを行うメソッドである．swing の JTable におけるソート機能は Java1.6 から実装された機能であるため，それ以前の開発ではソート機能は開発者が自ら実装する必要があった．したがって，図 3-3-3 の 2 つのソースコードは実際にライブラリ化された例であり，このようなメソッドクローンを今回の調査で検出することができた．

```
39 private static Converter stringConverter =
new Converter() {
40  public Short convert(Object o) {
41   return parseShort(((String) o));
42  }
43 };

（中略）

49 public Object convertFrom(Object in) {
50  if (!CNV.containsKey(in.getClass())) throw
new ConversionException("cannot convert type: "
51    + in.getClass().getName() + " to: " +
Short.class.getName());
52  return CNV.get(in.getClass()).convert(in);
53 }

（中略）

62   CNV.put(String.class,
63    stringConverter
64   );
```

(a) mvel　ShortCHクラス

```
17 public Object convertFrom(Object in) {
18  if (!CNV.containsKey(in.getClass())) throw
new ConversionException("cannot convert type: "
19    + in.getClass().getName() + " to: " +
Short.class.getName());
20  return CNV.get(in.getClass()).convert(in);
21 }

（中略）

30   CNV.put(String.class,
31    new Converter() {
32     public Short convert(Object o) {
33      return Short.parseShort(((String) o));
34     }
35    }
36   );
```

(b) mvflex　ShortCHクラス

図 3-3-2　流用特定の例

```
244 public void n2sort() {
245  for(int i = 0; i < getRowCount(); i++) {
246   for(int j = i+1; j < getRowCount(); j++) {
247    if (compare(indexes[i], indexes[j]) == -1) {
248     swap(i, j);
249    }
250   }
251  }
252 }
```

(a) sun　TableSorterクラス

```
219 public void n2sort() {
220  for (int i = 0; i < getRowCount(); i++) {
221   for (int j = i+1; j < getRowCount(); j++) {
222    if (compare(indexes[i], indexes[j]) == -1) {
223     swap(i, j);
224    }
225   }
226  }
227 }
```

(b) Perham　TableSorterクラス

図 3-3-3　ライブラリ化の例

④ 実験 2

次に，IT Spiral 実プロジェクトデータのソースコードに含まれるメソッドが，UCIdatasets に含まれるプロジェクトのメソッドと，メソッド単位でクローンになっているかどうかを確認した．メソッドクローン検出ツールの対象となる IT Spiral 実プロジェクトデータに含まれるメソッド数は 208 であった．結果として，クローンとして検出されたメソッド数は 17，クローンセット数は 10 であった．IT Spiral 実プロジェクトデータ内でのみのクローンメソッド数は 14，クローンセット数は 7 (要素数は全て 2)であった．また,プロジェクトをまたぐクローン(すなわち UCIdatasets のメソッドとクローンとなっているもの)のメソッド数は 3，クローンセット数は 3 であった．

プロジェクトをまたぐクローンの分析結果を述べる．上述の通り，プロジェクトをまたぐクローンセット数は 3 あった．先ず，クローンセット 1 は要素数が 2，またがっているプロジェクト数が 2 であり，それはセッションを扱うメソッドであった．クローンセット 2 は要素数が 2，またがっているプロジェクト数が 2 で，データベースの型を扱うメソッドであった．クローンセット 3 は要素数が 3，またがっているプロジェクト数が 3 で，文字のエンコーディングを扱うメソッドであった．いずれのクローンもライブラリ化には適さないと考えられる．またがっているプロジェクト数が小さい ため汎用的ではなく，処理内容が if 文+ return 文 であり実装コストが小さいからである．また，これらのメソッドは流用されたものではないと考えられる．クローンセット 1～3 の例をそれぞれ図 3-3-4，図 3-3-5，図 3-3-6 に示す．

```
public static Session currentSession() {
      Session s = SESSION.get();
      if (s == null) {
         s = SESSION_FACTORY.openSession();
         SESSION.set(s);
      }
      return s;
  }
```

IT Spiral実プロジェクトデータ：
HibernateUtil.java

```
private static Session session()
 {
    Session session = currentSession.get();

    if (session == null) {
       session = sessionFactory.openSession();
       currentSession.set(session);
    }

    return session;
 }
```

jmockit：Persistence.java

図 3-3-4　クローンセット 1 の例

```
public DataType createDataType(int sqlType, String sqlTypeName)
        throws DataTypeException {
    if (sqlType == Types.BOOLEAN) {
       return DataType.BOOLEAN;
    }

    return super.createDataType(sqlType, sqlTypeName);
 }
```

IT Spiral実プロジェクトデータ: : H2DataTypeFactory.java

```
public DataType createDataType(int sqlType, String sqlTypeName) throws DataTypeException {
    if (sqlType == Types.BOOLEAN) {
       return DataType.BOOLEAN;
    }

    return super.createDataType(sqlType, sqlTypeName);
 }
```

Mogwai Java Tools : HsqlDataTypeFactory.java

図 3-3-5　クローンセット 2 の例

```
public void doFilter(final ServletRequest request,
        final ServletResponse response, final FilterChain chain)
        throws IOException, ServletException {
    if (encoding != null) {
        request.setCharacterEncoding(encoding);
    }
    chain.doFilter(request, response);
  }
```

IT Spiral実プロジェクトデータ:
CharacterEncodingFilter.java

```
public void doFilter(ServletRequest request, ServletResponse response,
                FilterChain chain)
    throws IOException, ServletException
  {
    if (mEncoding != null)
    {
       request.setCharacterEncoding(mEncoding);
    }

    chain.doFilter(request, response);
  }
```

jamm (Java Mail Manager):
SetEncodingFilter.java

```
public void doFilter(ServletRequest req, ServletResponse res,
                        FilterChain chain) throws IOException, ServletException {
            if (encoding != null) {
                        req.setCharacterEncoding(encoding);
            }
                        chain.doFilter(req, res);
}
```

cosmos4j : EncodingFilter.java

図 3-3-6　クローンセット 3 の例

3.3.3 実用化へ向けた課題と問題点

(1) 課題と問題点

本テーマでは，大規模なソースコード群から，適切な大きさ（メソッド単位）のコードクローンを検出することで，再利用ライブラリに有用なものや違反流用コードを抽出する手法を提案した．

本テーマを実施するに当たって，想定する仮説は，「適切な大きさのコードクローン検出することで，再利用ライブラリに有用なものや違反流用コードを抽出できるか」ということになる．提案手法を用いたコードクローン検出ツールを実装し，オープンソースソフトウェアに対して検出を行った．結果として，再利用に有用なメソッド単位のコードクローンや流用されたメソッドを検出することができた．

今後の課題としては，本研究とは別の正規化やフィルタリング処理を実装して検出されるコードクローンに違いがあるかを調査すること，対象となるファイルを Java 以外にも拡張して，言語によって検出されるコードクローンに違いがあるかを調査することが考えられる．

(2) 将来の応用方法

今後の課題としては，約 4 億行のソースコードを含む大規模リポジトリより約 5 時間でメソッド単位のコードクローンを検出できていることより，企業で開発されたソースコード群に対しても十分なスケーラビリティで適用が可能であると考えている．適用対象言語が現状は Java だけであるが，最近のソフトウェア開発で Java が用いられるケースは非常におおいため，有用であると期待する．

４．考察

4.1 研究により判明した効果や問題点等

4.1.1 ソースコード理解支援

(1) 到達目標に対する達成度

本テーマでは，既存のコードクローン検出ツールと比較して，冗長なコードクローンをなるべく含まず，処理が高速で，検出したコードクローンの質が高い（再現率，適合率が高い）コードクローン検出ツールの開発を目指し,「どのようなコードクローンを冗長であると見なすかと言うことと，冗長でないコードクローンを高速に，かつ，再現率・適合率を高く検出できるプロトタイプシステムの開発を行う」ことを到達目標とした．

結果として，字句単位のコードクローン検出手法を対象に，冗長なコードクローンをソースコード中の同じ命令が繰り返し記述された部分（繰り返し部分）において，利用者にとって余計に検出されるコードクローンと考え，ソースコード上の繰り返し構造を折りたたむという検出の前処理を行った上でコードクローン検出する手法を提案した．また，プロトタイプシステムを開発し，評価実験を行った．実験の結果，折りたたみ処理を行う場合と行わない場合で．コードクローンの数が削減した（すなわち，冗長な部分が検出されなくなった)．また，検出されたコードクローンの質についても，適合率が全て改善したことが確認された．更に，高速な検出という点では，比較対象として想定した CCFinder に比べて開発したプロトタイプシステムの方が実験では検出時間が早いことを確認した.以上の結果から,設定した到達目標は十分に達成できたと考えている．

なお，実験結果の妥当性に関して，以下で挙げる点に留意する必要がある．本実験では，Bellon らが作成した正解クローンを用いて新正解クローンを作った．そして，新正解クローンを用いて，提案手法を実装したツールの性能について調査を行った．正解クローンは対象ソフトウェアに含まれるすべてのコードクローンではなく，それを用いて作成した新正解クローンも同様である．そのため，すべてのコードクローンを対象にして同様の実験を行った場合は，異なる実験結果が得られる可能性がある．しかし，ソースコード中に存在するすべてのクローンを対象にすることは現実的ではないので，正確な recall，precision の値を求めることは難しい．recall，precision の値は相対的な評価にのみ用いることができる．また，コードクローン検出ツールは字句や行単位のものだけでなく，抽象構文木やプログラム依存グラフを用いたものなど様々である．各コードクローン検出ツールによってコードクローンの定義が異なるので，他のツールを用いて比較をした場合，異なる実験結果が得られる可能性がある．

(2) 関連研究

文献[Higo2007]では，クローンペアの集合に含まれるコード片がどの程度繰り返し要素を含まないか表すメトリクス RNR を提案している．このメトリクスを使うことで関数や代入文などの羅列,switch 文のように同じ構造になりやすい文などを取り除くことが

できる.RNR は,CCFinder [Kamiya2002] の後続ツールである CCFinderX [CCFinderX] でも採用されており，閾値を定めることで検出結果のフィルタリングができる．RNR は，繰り返し要素から構成されるコードクローンをある程度自動的にフィルタリングすることは可能だが，把握すべきクローンを検出することができない点においては提案手法に対して劣っている．

今回の実験では，比較に Bellon らの実験を用いた．他にもコードクローン検出手法・ツールの比較実験は行われているが，Bellon らの実験は比較ツールの数，対象プログラムの規模共に最大であることから,本研究における比較実験として Bellon らの実験方法を採用した．他の関連として Burd らの実験[Burd2002]，Rysselberghe らの実験[Ryssel2004] を以下に示す．Burd らは Kamiya らの手法[Kamiya2002]，Baxter らの手法[Baxter1998]，Mayland らの手法[Mayland1996]，Prechalt らの手法[Prechelt2000]，Aiken らの手法[Moss]の 5 つの比較をしている．各検出手法で検出されたコードクローンを実際に見て，本当にコードクローンであったものを正解クローンとし，各検出手法の再現率と適合率を求めている．Rysselberghe らは，行単位，字句単位，メトリクス計測の 3 つの検出技法を比較している.Rysselberghe らはツールを比較するのではなく検出技法をするため，各検出技法を用いたツールを作成し，実験を行った．行単位，字句単位の検出は各プログラミング言語用に解析器を作る必要があるが，それほどコストは高くないと述べている．また，メトリクス計測による検出はソースコードから様々な情報を得なければならないので，解析器を作るコストは高いと述べている．

4.1.2 リファクタリング支援

(1) 到達目標に対する達成度

本テーマでは，コードクローンに対して，既存のリファクタリング支援より多くのリファクタリングパターンに対応できるツールの開発/集約方法のガイドラインの開発を行うことを目指し,「どのようなリファクタリングパターンを対象として支援を行うかを決めることと，対象とした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検出手法の提案とプロトタイプシステムの開発を行うこと」を到達目標とした．

結果として，Template Method パターンと呼ばれるリファクタリングパターンを対象とし，それが適用可能なコードクローンを，プログラム依存グラフを用いて検出する手法を提案した．更に，提案手法を用いたプロトタイプシステム（コードクローン検出ツール）を実装し,複数のソースコードに対して検出を行った.その結果,Template Method パターンを適用可能なコードクローンを検出でき，一部については実際にリファクタリングを正しく実施できた．以上の結果から，設定した到達目標は十分に達成できたと考えている．

(2) 関連研究

Juillerat らは抽象構文木(Abstract Syntax Tree，以降 AST) を用いて Template Method パターンの適用を自動化する手法を提案している[3]．Juillerat らの手法の特

長として，適用後のソースコード例を提示することができるという点と，実行に要する時間的，空間的コストが小さいという点が挙げられる．しかし，各頂点が表すコード片の文字列が一致するか否かで AST の頂点を比較しており，ユーザ定義名の違いを吸収することはできず，また AST を用いているため文の順序の違いや表現の違いを吸収することができないという課題点が存在する．

政井らは，Juillerat らの手法と同様に AST を用いて Template Method パターンの適用を支援する手法を提案している[4].AST から構築したトークン列をもとに比較を行う Juillerat らの手法と異なり，AST の構造的な情報を用いて比較を行っているため，機能的なまとまりのある差分を特定し，抽出することが可能である．また，適用可能な候補に対して，差異部分として抽出すべきコード片の候補を複数提示する機能や，パターンの適用容易性を判定する機能を備えている．しかし，ユーザ定義名の違いや表現の違い，文の順序の違い等の吸収には対応していない．

4.1.3 再利用ライブラリ作成支援・違反流用コード発見支援

(1) 到達目標に対する達成度

本テーマでは，大規模なソースコード群から，適切な大きさ（例えば，メソッド，関数単位）のコードクローンを検出することで，再利用ライブラリに有用なものや違反流用コードを抽出することを目指し,「特定のプログラミング言語で実装されたソースコード群を対象として，適切な大きさのコードクローンを検出するプロトタイプシステムを開発すること」を到達目標とした．

結果として，特定のプログラミング言語として Java を，適切な大きさとしてメソッド単位を設定し，メソッド単位のコードクローンを検出するプロトタイプシステムを実装した．更に，評価実験として，Java で開発されたオープンソースソフトウェアの大規模リポジトリを対象として，メソッド単位のコードクローンを検出し，再利用に有用なメソッド単位のコードクローン，違反流用されたと考えられるメソッド単位のコードクローンを検出できた．以上の結果から，設定した到達目標は十分に達成できたと考えている．

なお，実験結果の妥当性に関して，以下で挙げる点に留意する必要がある．先ず，メソッド単位のクローン検出で使っているハッシュ値の衝突である．提案手法では 2 つのメソッドがコードクローンであるかの判定にハッシュ値を使用している．そのためコードクローン検出の際に異なるメソッドのハッシュ値が偶然一致するハッシュ値の衝突が発生した場合，本来コードクローンでないメソッドがコードクローンとして検出されるおそれがある．しかし，実験で検出の対象となるメソッド数が約 600 万であり，かつハッシュ値の計算に 128bit の MD5 アルゴリズムを使用しているため衝突確率は極めて低い[Hummel2010].また,本実験では実際にハッシュ値の衝突が起こっているかどうかを，検出した全てのコードクローンに対して調査し，ハッシュ値の衝突がなかったことを確認した．ただし，多くのソフトウェアに適用していけば，いずれハッシュ値の衝突が起こる可能性がある．次に，正規化，フィルタリング処理の選択であるが，本提案では，3.3.2(2)②で述べた正規化やフィルタリング処理を行っている．しかし，型名の正規化

やメソッドサイズでのフィルタリングなど正規化やフィルタリング処理の方法を変更することによって本実験で得られた結果と異なる結果が得られる可能性がある．最後に，バージョンの異なる同一ソフトウェアの存在である．3.3.2(2)③で述べたように，UCIdatasets にはバージョンが異なる同一のソフトウェアが複数含まれている．本実験では，実験を行う前に可能な限り最新バージョンの以外のソフトウェアを解析対象から除外しているが完全ではなく，バージョンが異なる同一ソフトウェアが検出対象に含まれている可能性がある．そのため，不必要に多くコードクローンを検出している可能性がある．

(2) 関連研究

文献[Sasaki2011]では，ファイルクローン検出ツール FCFinder を開発し，ファイルクローンの性質を調査している．FCFinder はファイルをハッシュ値に変換しファイルクローンを検出する．また，FCFinder はコメント文の削除や字句解析を行うため，コメント文やインデントの違いを吸収できる．文献[Sasaki2011]では，総行数約 4 億行の大規模なソフトウェア群である FreeBSD Ports Collection に対し FCFinder を適用した結果，17 時間ほどで検出を終了し，FreeBSD Ports Collection の約 68%がファイルクローンであったことを報告している．また，検出されたファイルクローンのうち 27%はコメント文やインデントの違いであり，ファイルサイズの分布はファイルクローンであるファイルとそうでないファイルとで差異がなかったとも報告している．

文献[Ossher2011]では，ファイルクローン検出手法を提案し，ファイルクローンが発生する状況を調査している．文献[Ossher2011]の手法は，exact，FQN，fingerprint の 3 つの要素を組み合わせてファイルクローン検出を行う．exact は，各ソースファイルを 1 つの文字列とみなしその文字列をハッシュ値に変換し，ハッシュ値が一致したファイルをファイルクローンとして検出する．FQN は，クラスの完全限定名が一致しているファイルをファイルクローンとして検出する．fingerprint は，ソースファイル中のメソッド名とフィールド名がどの程度等しいかを調査し，ある閾値を超えて一致しているファイルをファイルクローンとして検出する．Ossher らは，約 1 万 3 千の Java ソフトウェアに対し上記の 3 つの要素を組み合わせて実験したところ，全ファイルの約 10%超がファイルクローンとして検出されたと報告している．またファイルクローンが発生する状況として，同じライブラリを使用していることや新たなソフトウェア開発を始めるときにそれ以前に開発されたソフトウェアの再利用を行うことなどが挙げられるとも報告している．

文献[Livieri 2007]では，分散処理技術を応用して大規模なソフトウェア群から短時間でコードクローンを検出する手法を提案し，その手法を D-CCFinder として実装した．D-CCFinder は既存のコードクローン検出ツール CCFinder に分散処理技術を加えている．対象とするソフトウェア群を一定のサイズで分割し，分割した各要素をそれぞれ別のコンピュータに割り振り，そこで CCFinder を実行する．Livieri らは，総行数約 4 億行の大規模なソフトウェア群である FreeBSD Ports Collection に対し D-CCFinder を適用した結果，CCFinder では 45 日を要すると予測されていたコードクローン検出処理が 51 時間で終了したと報告している．Livieri らの手法と本研究ではコードクローン検出の高速化を実現するアプローチが異なる．Livieri らは高コストである検出処理を分散させて多くの PC を利

用することで高速化を図っているが，我々は検出単位の粒度を大きくすることで検出コストを削減し，1 台の PC で高速化を実現している．

図 4-1 は，ソフトウェア間にまたがるコードクローンの例である．ハイライトされている 2 つのメソッドは木構造の表示のために使用される dump メソッドである．このメソッドを含むファイルは，ファイル全体としてはコードクローンにはなっていなかった．つまり，ファイルクローン検出手法ではこのメソッドをコードクローンと判断できない．本テーマで実施したメソッド単位のコードクローン検出手法では，図 4-1 のクローンを検出可能であり，従来手法に対する優位性を示している．

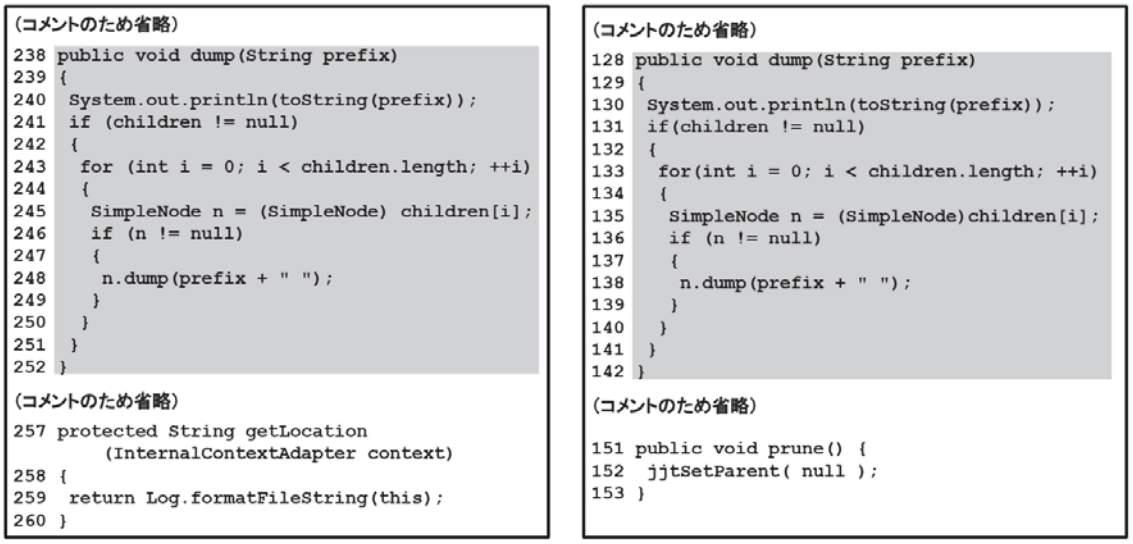


図 4-1　ファイル単位の検出手法では検出できないコードクローン

4.2 今後の課題

ここでは，4 つのテーマについて，今後の課題，研究予定をまとめる．

先ず，ソースコード理解支援については，検出対象の言語を増やすことが考えられる．本テーマでは，対象言語を C, Java とした．いわゆるレガシーシステムでは COBOL で開発されたものが未だ多いと考えられる．従って，実際の現場で数多く利用されている言語について本手法を拡張するということは一つの重要な課題である．また，より実用性を高めるためには，表示方法の改良など，ユーザインターフェースの充実も必要である．

次に，リファクタリング支援に関しては，Template Method パターン適用後のソースコード例の提示，及び 3 つ以上の類似メソッドへの拡張などが挙げられる．また，検出対象言語を増やすということも考えられる．どんな言語であっても，PDG の作成ができれば，提案手法の適用は容易に可能であると考えている．

最後に，再利用ライブラリ作成支援・違反流用コード発見支援であるが，Java 以外の言語への適用に関しては，例えば，関数単位や手続き単位でソースコードを分割できれば容易に適用が可能であるため，実装は比較的簡単であると考える．実務への適用として，ある特定の組織で開発したソースコード群に本手法を適用することで，その組織内で有用なメソッド単位のコードクローンの検出が見込まれる．また，多くのオープンソースソフト

ウェアをインターネット上から収集することで，オープンソースソフトウェアとの間の流用の検出が可能となる．

引き続き，本研究全体についての今後の課題をまとめる．

・コードクローン検出の実施時期

本研究では，ソフトウェア保守を対象としたため，開発完了後のソフトウェアに対する適用を行っている．コードクローンの作り込みを避けるためには，コードクローンが作り込まれた直後に，それを指摘するような仕組みが必要であるという意見もある．これについては，検出対象のソースコードの量にも依存するが，提案したツールはすべて開発途中においても適用可能であるので，開発工程の終了時点でコードクローン検出ツールを適用し，発見されたコードクローンに対するインタラクティブな対策をとるということは可能であると考えている．特に，アジャイルなソフトウェア開発プロセスではリファクタリングを奨励しているため，開発途中でのコードクローン検出と対策は受け入れられやすいと思われる．

・コードクローン検出・修正のメリットの提示

本研究ではコードクローンを開発・保守に悪影響を与える要因であるという前提で，様々な検出支援について研究を行った．実際の開発現場では，システムの残された不具合の中で，コードクローンを見直し，変更するときのコストや間違いなどのリスク，すべてを変更できない等といった状況が発生する．コードクローンの検出や修正には当然そのためのコストが必要である．従って，コードクローンの検出・修正を行うことによって，どの程度，将来の開発・保守作業に対するコストの削減が期待されるか，あるいは，品質がどの程度向上するかということについての，定量的な証左を提示することは非常に重要である．

この点に関しては，先ず，コードクローンの性質が非常に重要となる．実際の現場では，長年使われてきていて不具合も出てきておらず，信頼性の高いコードであれば積極的に流用することがある．この場合，コードクローンを作り込んだ（信頼性の高いコードを再利用した）ということが，ドキュメント等で記録され，将来の保守を通じて一貫性をもって対応されれば全く問題は無いと考える．

また，コードクローン対策を行うことによる，将来の開発・保守作業に対するコストの削減や品質の向上の定量的な評価については，企業における事例研究を実施する必要があると思われる．ある程度の期間，リビジョン管理がされているソフトウェアに対して，過去に作り込まれたコードクローンの履歴や実際に発見されたバグの位置情報，バグ修正に要したコスト等のデータを分析することで，あるコードクローンを一貫して修正，あるいは，集約していれば，そのバグが残されることがなかった，そのバグの発見修正に要したコストは削減できたという，評価が可能となる．オープンソースソフトウェアを対象としたコードクローンの履歴・変遷について分析した研究はあるが，コストや品質面の影響を分析した研究は我々の知る限りほとんどない．コスト，品質についての詳細データを記録している企業のデータを用いた事例研究を行うことは今後の大きな課題である．

・コードクローンの履歴情報

上記課題とも関連するが，コードクローンを，だれがいつ入れたのか，なぜ入ったのかが見えてくれば，対策も容易になると考えられる．これについても，ソースコード上に作成者名がコメントとして記入されている，ソースコードがリビジョン管理されており，リビジョン間の変更内容が記録されていれば，その情報を基に，コードクローンの詳細履歴情報を収集し，分析することは可能である．そのためには，上記の情報を収集するための基盤が確立されている必要がある．

・マルチベンダ開発における活用

大規模システム開発においては，複数のベンダが参画することが多く，参画しているソフトウェア会社によって品質の違いが出ることも多い．各社のソースコードのモジュールのタイムスタンプを見て，「深夜にタイムスタンプがあるものはクローン率が高い」といったような仮説を立てて，コードクローンとベンダ間の品質や生産性の評価を行うということも考えられる．そのためには，マルチベンダを束ねる組織のリーダーシップやコードクローン分析に対する現場の理解を確立する必要がある．

参考文献

[Keisan] 経済産業省・特定サービス産業実態調査 (http://www.meti.go.jp)

[Soumu] 総務省情報通信政策局「通信利用動向調査報告書世帯編」

[Nikkei] 日経コンピュータ 2003 年 11 月 17 日号, 2008 年 12 月 11 日号.

[Tokkyo] http://www.jpo.go.jp/

[ICSE] http://www.icse-conferences.org/

[ICSM] http://conferences.computer.org/icsm/

[Douyama2007] 堂山真一: “Java ソースのコードクローンと品質改善活動～NTT コムウェアにおける取り組み事例～”, ソフトウェア工学工房セミナー (2007).

[Kamiya2002] Toshihiro Kamiya, Shinji Kusumoto and Katsuro 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).

[Higo2004] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: ″Refactoring Support Based on Code Clone Analysis″, Proceedings of the 5th International Conference on Product Focused Software Process Improvement(Profes 2004), pp.220-233 (2004).

[Morisaki2008] 森崎修司, 吉田則裕, 肥後芳樹, 楠本真二, 井上克郎, 佐々木健介, 村上浩二, 松井恭: “コードクローン検索による類似不具合検出の実証的評価”, 電子情報通信学会論文誌 D, Vol.J91-D, No.10, pp. 2466-2477(2008).

[Li2006] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou: “CP-Miner: finding copy-paste and related bugs in large-scale software code”, IEEE Transactions on Software Engineering, Vol.32, No.3, pp. 176-192 (2006).

[Blackduck] http://www.blackducksoftware.com/jp/

[Bellon2007] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, Ettore Merlo: “Comparison and Evaluation of Clone Detection Tools”, IEEE Transactions on Software Engineering, Vol. 33, No.9, pp. 577-591 (2007).

[Fowler1999] Martin Fowler: Refactoring: Improving the Design of Existing Code, Addison Wesley (1999).

[Higo2005] 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎 :“コードクローンを対象としたリファクタリング支援環境”, 電子情報通信学会論文誌 D, Vol. J88-D-I, No.2, pp. 186-195 (2005).

[Higo2007] 肥後芳樹, 吉田則裕, 楠本真二, 井上克郎:“産学連携に基づいたコードクローン可視化手法の改良と実装”, 情報処理学会論文誌, Vol.48, No.2, pp.811-822 (2007).

[Natsuura2006] 松浦清, 神谷芳樹, 樋口登： “先進ソフトウェア開発プロジェクト PartII”, SEC journal, Vol.5, pp.44-49 (2006).

[German2010] Daniel M. German, Yuki Manabe, and Katsuro Inoue: “A Sentence-Matching Method for Automatic License Identification of Source Code Files”, Proceedings of the 25th Automated Software Engineering, pp. 437-446 (2010).

[Ren2004] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder and Ophelia Chesley: “Chianti: A tool for change impact analysis of Java programs”, Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), pp. 432-448 (2004).

[Ryder2001] Barbara G. Ryder and Frank Tip: “Change impact analysis for object-oriented programs”, Proceedings of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE 2001), pp. 46-53(2001).

[Kataoka2002] Yoshio Kataoka, Takeo Imai, Hiroki Andou and Tetsuji Fukaya: “A quantitative evaluation of maintainability enhancement by refactoring”, Proceedings of the 18th International Conference on Software Maintenance, pp.576-585(2002).

[Hatano2003] 秦野克彦, 乃村能成, 谷口秀夫, 牛島和夫: “ソフトウェアメトリクスを利用したリファクタリングの自動化支援機構”, 情報処理学会論文誌, Vol. 44, No.6, pp.1548-1557 (2003).

[McMillan2011] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie and Chen Fu: “Portfolio: finding relevant functions and their usage”, Proceedings of the 33rd International Conference on Software Engineering, pp. 111-120 (2011)

[Zhao2004] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun and Fuqing Yang: “SNIAFL: Towards a Static Non-Interactive Approach to Feature Location”, Proceedings of the 26th International Conference on Software Engineering, pp. 293-303(2004).

[Palamidia] Palamidia, http://www.palamida.com

[Ueda2002a] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue: “Gemini: Maintenance Support Environment Based on Code Clone Analysis”, Proceedings of the Eighth IEEE Symposium on Software Metrics, pp. 67-76(2002).

[Ueda2002b] Yasushi Ueda, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue: “Gemini: Code Clone Analysis Tool”, Proceedings of 2002 International Symposium on Empirical Software Engineering, Vol.2, pp.31-32 (2002).

[Ueda2002c] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue:

“On Detection of Gapped Code Clones using Gap Locations”, Proceedings of the 9th Asia Pacific Software Engineering Conference, pp.327-336 (2002).

[Ueda2003] 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 D-I, Vol. J86-D-I, No. 12, pp. 863-871 (2003).

[Izumida2003] 泉田聡介, 植田泰士, 神谷年洋, 楠本真二, 井上克郎:“ソフトウェア保守のための類似コード検索ツール”, 電子情報通信学会論文誌 D-I, Vol.J86-D-I, No.12, pp.906-908(2003).

[Higo2010] 肥後芳樹, 楠本真二, ″プログラム依存グラフを用いたコードクローン検出法の改善と評価,″ 情報処理学会論文誌, Vol. 51, No. 12, pp. 2149-2168 (2010).

[Lopes] Cristina Lopes, Sushil Bajracharya, Joel Ossher and Pierre Baldi: Ucisource code data sets. http://www.ics.uci.edu/~lopes/datasets/.

[Higo2008] 肥後芳樹, 楠本真二, 井上克郎: “コードクローン検出とその関連技術”, 電子情報通信学会論文誌 D, Vol. J91-D, No. 6, pp.1465-1481 (2008).

[Simian] http://www.harukizaemon.com/simian/

[Juillerat2007] Nicolas Juillerat and Beat Hirsbrunner: “Toward an Implementation of the “Form Template Method” Refactoring”, Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation, pp.81-90 (2007).

[Masai2010] 政井智雄, 吉田則裕, 松下誠, 井上克郎：“類似メソッド集約のための差分抽出支援”, 電子情報通信学会技術報告, Vol.110, No.60, pp.45-50 (2010).

[Gamma1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional (1995).

[Ferrante1987] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren: “The Program Dependence Graph and Its Use in Optimization”, ACM Transactions on Programming Languages and Systems, Vol.9, No.3 , pp. 319-349 (1987).

[Miyake2009] 三宅達也, 肥後芳樹, 楠本真二, 井上克郎：“多言語対応メトリクス計測プラグイン開発基盤 MASU の開発”, 電子情報通信学会論文誌 D, Vol. J92-D, No. 9, pp. 1518-1531(2009).

[Sasaki2011] 佐々木裕介, 山本哲男, 早瀬康裕, 井上克郎：“大規模ソフトウェアシステムを対象としたファイルクローンの検出”, 電子情報通信学会論文誌 D, Vol.J94-D, No.8, pp.1423-1433(2011).

[Ossher2011] Joel Ossher, Hitesh Sajnani and Cristina Lopes: “File cloning in

open source java projects: The good, the bad, and the ugly”, Proceedings of the 27th International Conference on Software Maintenance, pp.283-292 (2011).

[MD5] Ronald L. Rivest: “The md5 message-digest algorithm”, (1992). http://www.ietf.org/rfc/rfc1321.txt.

[CCFinderX] http://www.ccfinder.net/ccfinderx-j.html.

[Burd2002] Elizabeth Burd and John Bailey: “Evaluating clone detection tools for use during preventative maintenance”, Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation, pp. 36-43(2002).

[Ryssel2004] Filip Van Rysselberghe and Serge Demeyer: “Evaluating clone detection techniques from a refactoring perspective”, Proceeding of the 19th IEEE International Conference on Automated Software Engineering, pp. 336-339(2004).

[Baxter1998] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna and Lorraine Bier: “Clone detection using abstract syntax trees”, Proceedings of the 14th I International Conference on Software Maintenance, pp. 368-377 (1998).

[Mayland1996] Jean Mayrand, Claude Leblanc aqnd Ettore Merlo: “Experiment on the automatic detection of function clones in a software system using metrics”, Proceedings of the 12th I International Conference on Software Maintenance, pp. 244-253(1996).

[Prechelt2000] Lutz Prechelt, Guido Malpohl and Michael Philippsen: “Jplag: Finding Plagiarisms among a Set of Programs with JPlag”, Journal of Universal Computer Science, Vol.8, pp.1016-1038(2000).

[Moss] Moss: A system for detecting software plagiarism. http://www.cs.berkeley.edu/~aiken/moss.html.

[Hummel2010] Benjamin Hummel, Elmar Juergens, Lars Heinemann and Michael Conradt: “Index-based code clone detection: incremental, distributed, scalable”, Proceedings of the 26th International Conference on Software Maintenance, pp. 1-9 (2010).

[Livieri 2007] Simone Livieri, Yoshiki Higo, Makoto Matushita and Katsuro Inoue: “Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder D-ccfinder”, Proceedings of the 29th International Conference on Software Engineering, pp. 106-115 (2007).