社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

メトリクス計測や解析のためのソースコード平坦化

肥後　芳樹†　　楠本　真二†

† 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　〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソースコードに対してメトリクスの計測や解析を行う場合には，その前処理として正規化が行われることが多い．正規化を行うことよりソースコード中から処理の対象ではない要素が取り除かれたり，ソースコード中の要素が整形されたりするため，より有意な処理結果が得られる．既存の正規化は，プログラムの振る舞いに影響を与えないコメントや空白行の削除や整形，もしくは変数名やリテラルを一定のルールに従って他の名前に置換するというプログラムの字句が対象のものがほとんどである．本研究では，複雑なプログラム文や条件式を平易なプログラム要素へと変換する正規化を提案する．本研究ではこの正規化を平坦化と呼ぶ．平坦化によって，平易なプログラム要素と複雑なプログラム要素が入り混じったソースコードは，平易なプログラム要素からのみ構成されるソースコードへと変換される．本稿では平坦化の効果を調査するために，平坦化前後のソースコードに対する行数メトリクスの計測結果およびコードクローンの検出結果についても報告する．
キーワード　ソースコード解析，メトリクス計測，ソースコードの正規化

1. はじめに

ソフトウェアのソースコードはソフトウェア工学における主要な研究対象の 1 つであり，さまざまな目的のためにソースコードを対象とした研究が行われている．

- ソースコードの解析を行う目的の 1 つとして，現状のソースコードの把握が挙げられる．ソースコードの規模や複雑度，その構造に関するデータを定量化して開発者に提示することでソースコードの状態をより客観的に把握できる [8]．また，関数・メソッドの呼び出し関係やオブジェクト指向言語におけるクラスの継承関係等，ソフトウェアを構成するモジュール間の依存関係を可視化する研究も行われている [13], [22].
- 既存のソフトウェア資産を効果的および効率的に再利用することを目的とした研究においてもソースコードは解析対象である．頻繁に再利用されるソースコード部品やソースコード中に頻発するコードのパターンの特定 [1] やコーディング中にその場所で利用可能な関数・メソッド等の補完を行う研究が行われている [20].
- ソフトウェアの問題点を発見したり改善案を提示したりする研究も広く行われている．問題点を発見する研究としてはフォールトプローンモジュールや脆弱性を含むコードの特定 [5], [15]，改善案を提示する研究としてはリファクタリングを行うべきコードの特定 [17] が挙げられる．

それらの研究において広く利用されている技術としてメトリクスの計測や頻出コードパターンの検出が挙げられる．モジュールの行数，循環的複雑度，CK メトリクス等がよく利用されるメトリクスである．頻出コードパターンの検出としては，似ているファイルや関数・メソッド，コード片の検出 [19] や関数やメソッドの呼び出しパターンの検出が挙げられる．

メトリクスの計測や頻出コードパターンの検出では，より良い結果を得るために，計測処理や頻出コードパターンの検出処理に先立ってソースコードの正規化が行われることが多い．ここでの正規化とは，ある一定の規則に従ってソースコードを変換する処理のことである．例えば，メトリクスの計測においては，ソースファイル内のコメントを削除やフォーマットを統一といった正規化が行われる．頻出コードパターンの検出では，変数名やリテラルを特殊な文字に置換した上で検出の処理が行われる場合が多い．このように，既存の正規化方法はプログラムの振る舞いに影響を与えない空白行やコメントに関するものやプログラムの字句に対して行われるものが多い．

本研究では，ソースコードの構造を正規化する手法を提案する．提案手法により，ソースコード内に存在している複雑なプログラム文や条件式は複数の簡単なプログラム要素に分解される．つまり，複雑なプログラム文と平易なプログラム文が入り混じったソースコードは平易なプログラム文のみで構成されるソースコードに変換される．本研究ではこの正規化を平坦化と呼ぶ．平坦化により，ソースコード一行当たりのプログラムの機能が均一化されるため，行数等のメトリクスがより有意な値になることが期待される．また，構造が正規化されることにより，より多くの振る舞いが同一なコードが頻出コードパターンとして検出されることが期待される．

本稿では，ソースコードの平坦化手法の提案に加えて，平坦化したコードに対する行数メトリクスの計測と，コードクロー

ン検出の適用結果についても述べる．行数メトリクスの計測では，平坦化により大きく行数が増えるソースファイルが多数あることが確認された．また，ファイルの行数メトリクスとそのファイルに存在しているバグの有無の相関を調査したところ，提案手法を適用することにより，多くの場合において相関が高くなることが確認された．コードクローン検出については，提案手法を適用することでより多くのコードクローンが検出されるようになることが確認された．

2. 関連研究

ソースコードの正規化はコードクローンの検出手法でよく利用される [3], [7], [10]〜[12], [18], [21] ．コードクローン検出ではソースコードの正規化は変数名を特殊文字に置換することが多い．全ての変数を同一の特殊文字に置換する方法と，異なる変数名を異なる特殊文字に置換する方法がある．後者は変数名の対応付けが保たれているので，前者の正規化に比べて誤検出が少ない．

ソースコードの解析を支援する方法として，ソースコードを解析しやすい別の形式に変換するという研究も行われている．Cordy は TXL というソースコードを変換するためのフレームワークを提案している [6]．Maletic らは XML でソースコードを表現するフレームワーク srcML を提案している [16]．ソースコードを XML で表現することで，XML を操作するさまざまな既存ツールを利用することでソースコードを操作できる．

ソースコードの構造を変更する手法としては，ソースコード中から goto 文を取り除き振る舞いが等価なプログラムを生成する手法が提案されている [2], [23]．これらの手法には変換の際に制御依存グラフを利用する．制御依存グラフの形状が変わらないソースコードが出力される．

ソースコードに対して自然言語処理の手法を適用する際には，ユーザ定義名の正規化（変数名等を英単語に分割）が行われる [14]．また，マルウェアを検出するためのアセンブリコードの正規化方法も提案されている [4]．

3. 研究の動機

図 1 は，あるオープンソースソフトウェアのソースコードの一部である．このように，ソースコードには簡単なプログラム文のみからなるコード (例えば，259 行目) や複雑な式 (例えば，269 行目) を持つコードが存在する．このような簡単なコードと複雑なコードの混在は，メトリクス計測やソースコード解析に悪影響を与える恐れがある．

メトリクス計測の例として行数を計測する場合を考える．行数はソースコードの規模を表すメトリクスである．より有意な行数を計測するために，空白行やコメント等を除外した行数が計測されることが多い．行数メトリクスを，そのソースファイルが含んでいる機能の量やフォールトプローンモジュールの特定の際の指標にする場合，簡単なコードと複雑なコードの混在は，指標としての精度を劣化させる要因となりうる．例えば，複雑なコード 50 行と簡単なコード 100 行ではどちらが機能的

```
…
256  public int read() throws IOException {
257    if (needAddSeparator) {
258      if (lastPos >= eolString.length()) {
259        lastPos = 0;
260        needAddSeparator = false;
261      } else {
262        return eolString.charAt(lastPos++);
263      }
264    }
265    while (getReader() != null) {
266      int ch = getReader().read();
267      if (ch == -1) {
268        nextReader();
269        if (isFixLastLine() && isMissingEndOfLine()) {
270          needAddSeparator = true;
271          lastPos = 1;
272          return eolString.charAt(0);
273        }
274      } else {
275          addLastChar((char) ch);
276          return ch;
277      }
278    }
279    return -1;
280  }
…
```

図 1　複雑なプログラム文を持つソースコード

```
…
284        needAddSeparator = false;
285      } else {
286        int $24 = lastPos++;
287        char $25 = eolString.charAt($24);
288        return $25;
289      }
…
295      if (ch == -1) {
296        nextReader();
297        boolean $28 = isFixLastLine();
298        boolean $29 = isMissingEndOfLine();
299        if ($28 && $29) {
300          needAddSeparator = true;
301          lastPos = 1;
…
```

図 2　図 1 のソースコードを提案手法により平坦化した後の状態

に多いかを知ることは難しい．また，50 行と 100 行という情報だけでは，後者の方をよりフォールトプローンであると判定してしまうだろう．

ソースコード解析の例としてコードクローンの検出を挙げる．検出されたコードクローンは，類似バグを含むコードの特定やコードの集約等に利用される．このように，コードクローンの検出は同様の機能を持つコードの特定の手段として利用されることが多い．さまざまなコードクローン検出の方法が提案されているが，よく利用されるのは字句単位の検出や抽象構文木を利用した検出である．字句単位の検出では，字句解析によってソースコードは字句の列へと変換される．そして連続して一致する字句の部分列がコードクローンとして検出される．抽象構文木を利用した検出では，構文解析によってソースコードから抽象構文木が構築される．そして，同形の部分木がコードクローンとして検出される．ソースコード中に機能的に同一な 2 つのコード片があるとする．片方は，簡単なプログラム文のみで構成されている．もう片方は複雑な式が入り組んだ構造を持っているとする．ことような場合，既存のコードクローン検出手法では，それらをコードクローンとして検出することは難しい．

そこで本研究では，複雑な文や式を複数の簡単な文へと分解

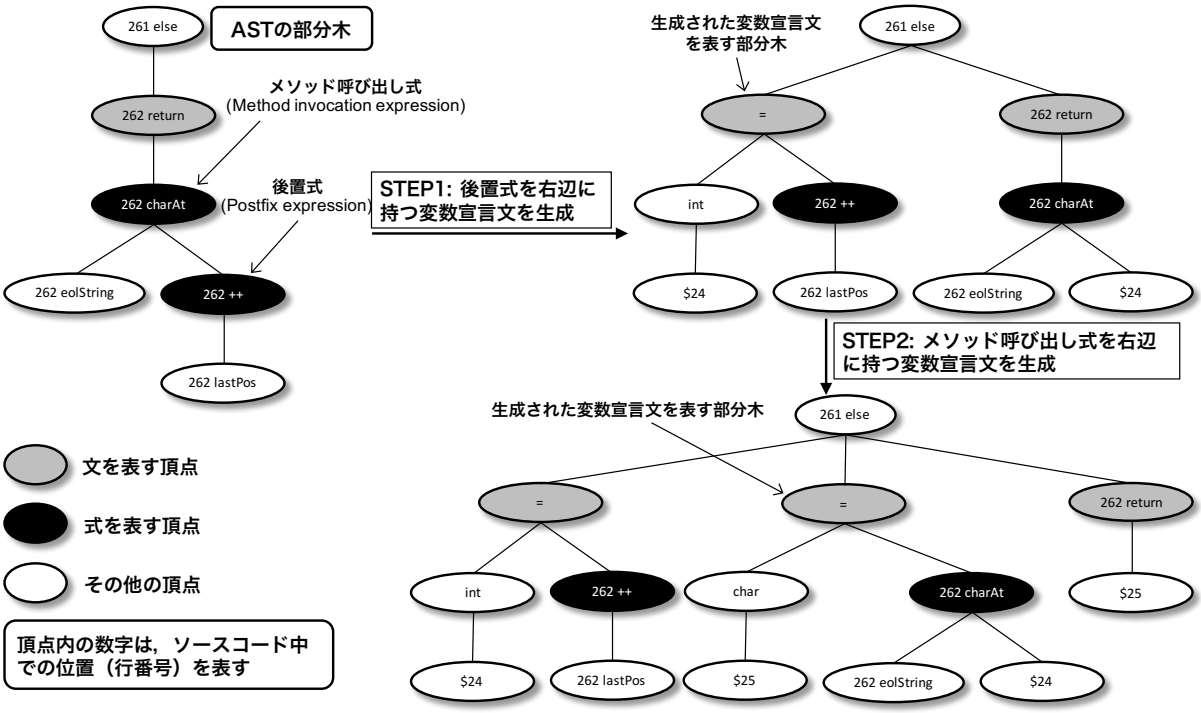

図 3　提案手法による AST 変換処理の様子

する手法を提案する．本研究ではこの分解を平坦化と呼ぶ．平坦化によって，ソースコード中から複雑に入り組んだ文や式はなくなる．そのため，一行あたりの機能量が均一化され，行数メトリクスの値がより有意になることが期待される．また，振る舞いが同一なコードがより類似した構造を持つようになるため，それらをコードクローン検出手法を利用して見つけやすくなると期待される．

図 2 は，図 1 のソースコードを提案手法により変換したものである．元のソースコードの 262 行目が変換後の 286~288，269 行目が変換後の 297~299 に該当する．複雑な文や式が複数の簡単な文や式に分解されているのがわかる．

4. 提 案 手 法

ここではソースコードの平坦化手法を提案する．提案手法は抽象構文木（以降，AST）の実装に強く依存するため，本論文では Java のソースコード解析ツール Java Development Tools（以降，JDT）が構築するを対象として提案手法の説明を行う．

平坦化は AST を操作することにより行う．AST に存在している対象の部分木を走査し，平坦化条件が成り立つかどうかを判定する．平坦化条件が成り立つ場合には，抽出対象の部分木を抽出して，その部分木を右辺として持つ変数宣言文の部分木を生成する．抽出対象の部分木は，生成した変数宣言文において宣言された変数の参照に置換する．なお，提案手法により追加された変数宣言文において宣言された変数は，その右辺の式の型を利用する．これにより，平坦化後のコードはコンパイルおよび実行可能なコードとなる．

JDT の AST では，以下の頂点が平坦化の候補となる部分木である．

- 文を表す頂点
- 分岐やループの条件式を表す頂点
- for 文の初期化式，更新式を表す頂点

JDT の AST では，以下を除いた式を表す頂点が抽出対象である（表 1 は抽出対象の頂点に対応する JDT 内のクラスの一覧である）．

- 変数の参照を表す式
- 数値や文字等のリテラルを表す式
- null を表す式

図 3 は，提案手法による平坦化の様子を表している．この図の左上の AST は図 1 の 261 行目および 262 行目のコードを表している．部分木の操作は深さ優先探索の後順にて行われる．この例の場合は，後置式が抽出対象の部分木の根頂点として判定される．そのため，この後置式を右辺に持つ変数宣言文を AST に追加する．そして，抽出された箇所には追加された変数宣言文において宣言された変数の参照が置かれる．後置式の抽出後，再度この AST は走査され，メソッド呼び出し式が抽出対象の部分木として判定される．後置式と同様の処理が行われ，メソッド呼び出し式を右辺として持つ変数宣言文が追加される．この処理を抽出対象の式が見つからなくなるまで繰り返す．

図 4 は，提案手法を利用した別の変換例である．この図の左

表 1　JDT における抽出対象の AST 頂点

ArrayAccess	ArrayCreation
ArrayInitializer	CastExpression
ClassInstanceCreation	ConstructorInvocation
ExpressionMethodReference	InfixExpresssion
InstanceofExpression	MethodInvocation
PostfixExpression	PrefixExpression
SuperConstructorInvocation	SuperMethodInvocation

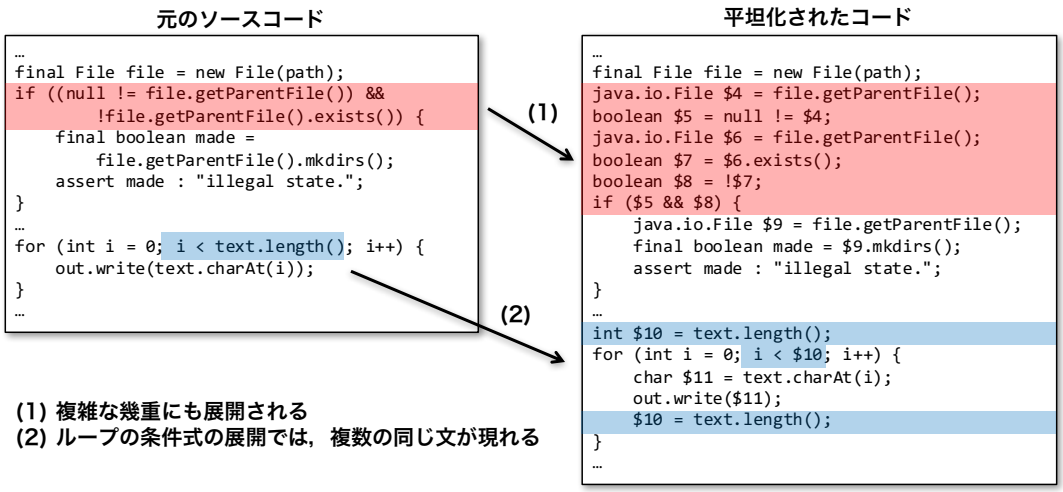

図 4　実装したツールを用いた変換の例

側は平坦化前のコードを表している．if 文の条件式が非常に複雑であり，その条件式から 5 つの変数宣言分が抽出されていることがわかる．このように提案手法は，対象の式が複雑であればあるほど，幾重にもその中に存在している式を変数宣言文として抽出する．

この図では for 文の条件式も平坦化の対象である．for 文のようなループの条件式の抽出では，ループの前に変数宣言文が追加され，そのループの最後に同じ変数への代入文(注1)が追加される．ループの条件式をこのように平坦化することにより，平坦化前のコードと実行結果が同じになることが期待できる．

5. 適 用 実 験

提案手法の適用実験として以下の 2 つを行った．

実験 A　平坦化によって行数メトリクスの計測結果およびコードクローンの検出結果がどのように変化するのかを調査．

実験 B　平坦化によって行数メトリクスとバグとの相関の度合いが程度変化するのかを調査．

以降，各実験について詳細に述べる．

5.1 実　験　A

実験 A では，著者らが過去の研究 [9] で構築したデータセットを利用した．このデータセットは，2 つのパッケージからなり，各パッケージにはそれぞれ 84 と 500 のソフトウェアのソースコードが含まれている．表 2 により詳細な規模のデータを載せる．このデータセットでは，テストコードやコンパイラコンパイラ等による自動生成コードが手作業により注意深く取り除かれている．

表 2　実験 A の対象プロジェクト

パッケージ	APACHE	UCI
プロジェクト数	84	500
ファイル数	66,724	60,548
総行数	11,545,556	10,073,635

(注1)：厳密には，Java および JDT には代入文という概念はない．ここでは代入演算子 "=" を持つ二項演算式を含む式文を，便宜上代入文と呼ぶ．

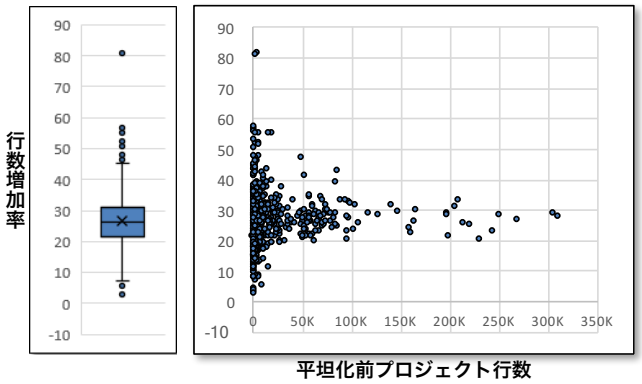

図 5　平坦化によるプロジェクト単位での行数の変化の様子

図 5 はプロジェクト単位での行数の変化の様子を表している．この図の縦軸に記載されている行数増加率は以下の計算式で算出される．なお，LOC_{ori}，LOC_{fla} はそれぞれ元のコードの行数，平坦化後のコードの行数を表す．

$$\text{行数増加率} = 100 * \frac{(LOC_{fla} - LOC_{ori})}{LOC_{ori}}$$

この図より，プロジェクトが小規模であるほど，平坦化により増加する行数の割合のばらつきが大きいことを表す．元のコードが 25,000 行以下の場合では，平坦化により行数が 1.5 倍以上になったプロジェクトも幾つか存在している．その一方で，プロジェクトの規模が 100,000 行を超えるような場合では，行数変化率は 20~35%程度の範囲に収まっている．この結果から，プロジェクトの規模が小規模であるほど一行あたりの密度にばらつきが大きく，行数の値を意思決定の基準としないほうが良いことがわかる．

図 6 はプロジェクト単位でのコードクローン検出数の変化の様子を表している．この図の縦軸に記載されているクローン検出数増加率は以下の計算式で算出される．なお，$CLONE_{ori}$，$CLONE_{fla}$ はそれぞれ元のコードから検出されたコードクローンの数，平坦化後のコードから検出されたコードクローンの数を表す．

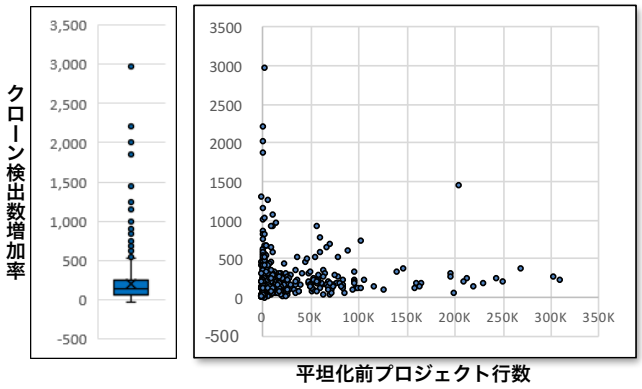

図 6　平坦化によるプロジェクト単位でのクローン検出数の変化の様子

$$\text{クローン検出数増加率} = 100 * \frac{(CLONE_{fla} - CLONE_{ori})}{CLONE_{ori}}$$

この図より，行数と同様に，プロジェクトの規模が小さいほど平坦化により新たに検出されるコードクローンの数にばらつきがあることがわかる．20,000 行以下のプロジェクトでは，10 倍以上のコードクローンが検出される場合もあった．その一方で，プロジェクトのサイズが 100,000 行を超える場合にはほとんどの場合において，クローンの検出量は 5 倍以内で収まっている．このことから，平坦化を行うことによって非常に多くのコードクローンが見つかるようになることがわかる．

図 7 は，平坦化によって新たに検出されたコードクローンを表している．146~158 行のコード片のうち，150 行目と 151 行目は元のソースコードでは 1 つの文であった．それに対して，300~312 行のコード片の 304 行目および 305 行目は元のソースコードでもこの状態であった．この差異により，元のソースコードでは２つのコード片はコードクローンとして検出されていなかった．平坦化を行うことにより，この実装方法の異なるコード片が既存のクローン検出手法を利用することで検出できた．

5.2 実　験　B

実験 B では Timmermann らの研究 [24] で用いられたバグのデータセットを利用した．このデータセットは彼らのウェブサイトで公開されている(注2) このバグデータセットの対象は，Eclipse のバージョン 2.0，2.1，3.0 のソースコードであり，それらは別途ダウンロードする必要がある(注3)．

実験は以下の手順で行った．ダウンロードしたソースコードに対して平坦化を行った．平坦化する前の行数と平坦化した後の行数をバグデータセットのメトリクスに追加した(注4)．付属の R スクリプトを実行し，得られた数値を確認した．数値は，見つかったバグ数の降順で並び替えられたファイルと，行数の降順で並び替えられたファイルとの間の Spearman の順位相関係数の値である．結果を表 3 に示す．なお，リリース前バグとはリリース日を終点とする 6 ヶ月間に報告されたバグであり，リリース後バグとはリリース日を起点とする 6 ヶ月間に報告されたバグを表す．なお，表示文字列の綴りが間違っている等のささいなバグは除外されている．ソースコード種別の「リリース版」とは，Eclipse のウェブページで公開されているソースコードをそのまま用いたことを意味している．「書式統一」とは，コメントの削除およびフォーマットの統一を行ったソースコードを意味している．「書式統一 + 平坦化」とは更に提案手法により平坦化を行ったソースコードである．

(注2)：https://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

(注3)：http://archive.eclipse.org/eclipse/downloads/

(注4)：バグデータセットのメトリクスとは，全ての対象ファイルについて，発見されたバグの個数と種々のメトリクスの値が保存されている CSV 形式のデータである．

```
    …
145 public static synchronized void closeTerminal(PosScreen pos) {
146   if (!mgrLoggedIn) {
147     pos.showDialog(“dialog/error/mgrnotloggedin”);
148     return;
149   }
150   Object $23 = pos.getSession();
151   PosTransaction trans = PosTransaction.getCurrentTx($23);
152   Object $24 = trans.isOpen();
153   if (!$24) {
154     pos.showDialog(“dialog/error/terminalclosed”);
155     return;
156   }
157   Output output = pos.getOutput();
158   Input input = pos.getInput();
159   if (input.isFunctionSet(“CLOSE”)) {
    …
299 public static synchronized void voidOrder(PosScreen pos) {
300   if (!mgrLoggedIn) {
301     pos.showDialog(“dialog/error/mgrnotloggedin”);
302     return;
303   }
304   XuiSession session = pos.getSession();
305   PosTransaction trans = PosTransaction.getCurrentTx(session);
306   Object $79 = trans.isOpen();
307   if (!$79) {
308     pos.showDialog(“dialog/error/terminalclosed”);
309     return;
310   }
311   Output output = pos.getOutput();
312   Input input = pos.getInput();
313   boolean lookup = false;
    …
```

コードクローンとして検出

図 7　平坦化により新しく検出されたコードクローン

6. おわりに

本稿では，ソースコード中の複雑なプログラム文や条件式を

表 3　JDT における抽出対象の AST 頂点

バージョン	ソースコード種別	リリース前バグ	リリース後バグ
2.0	リリース版	0.3873	0.3481
	書式統一	0.3938	0.3566
	書式統一 + 平坦化	0.3994	0.3623
2.1	リリース版	0.4216	0.2514
	書式統一	0.4206	0.2516
	書式統一 + 平坦化	0.4260	0.2540
3.0	リリース版	0.4227	0.3401
	書式統一	0.4235	0.3391
	書式統一 + 平坦化	0.4255	0.3396

複数の簡易なプログラム文に分解する手法を提案した．提案手法を適用することで，ソースコード中一行あたりの機能の量が均一化されるため，行数メトリクス等の値がより優位になることが期待される．また，ソースコードの構造がある程度統一されるため，そのままのソースコードからは見つけることが難しかったコードクローンを見つけることができるようになると期待される．

提案手法はツールとして実装され，著者のウェブページにて公開されている(注5)．作成したツールを用いて実験を行い，提案手法を評価した．実験の結果，提案手法を適用することで大幅に行数メトリクスの値が増加するソースファイルが多数あることが確認できた．また，提案手法を用いることでファイルの行数メトリクスの値とそのファイルに存在しているバグの有無がより高い相関関係を保つ場合が多いことが確認された．さらに，提案手法を適用したソースコードからはより多くのコードクローンが検出されることを確認した．

今後の課題としては，提案手法を適用することによって生成されたプログラム文のうち，同一のものが多数ある場合はそれらを 1 つにまとめることが挙げられる．このようなリファクタリングをすることによって，ソースコードがより簡潔な保守しやすい構造になる．

謝　　辞

本研究は，科学研究費補助金基盤研究 (S)(課題番号：25220003) および基盤研究 (B)(課題番号：17H01725) の助成を得て行われた．

文　　献

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 25–34, 2007.

[2] Z. Ammarguellat. A Control-Flow Normalization Algorithm and Its Complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, Mar. 1992.

[3] H. A. Basit, S. J. Puglisi, W. F. Smyth, A. Turpin, and S. Jarzabek. Efficient Token Based Clone Detection with Flexible Tokenization. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 513–516, 2007.

[4] D. Bruschi, L. Martignoni, and M. Monga. Code Normalization for Self-Mutating Malware. *IEEE Security and Privacy*, 5(2):46–54, Mar. 2007.

[5] C. Catal and B. Diri. A Systematic Review of Software Fault Prediction Studies. *Expert Systems with Applications*, 36(4):7346–7354, 2009.

[6] J. R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, Aug. 2006.

[7] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching: Research articles. *Journal of Software Maintenance and Evolution*, 18(1):37–58, Jan. 2006.

[8] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. PWS Publishing Co., Boston, MA, USA, 3rd edition, 2014.

[9] Y. Higo and S. Kusumoto. How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 294–305, 2014.

[10] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. ndex-based Code Clone Detection: Incremental, Distributed, Scalable. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–9, 2010.

[11] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective – A Workbench for Clone Detection Research. In *Proceedings of the 31st International Conference on Software Engineering*, pages 603–606, 2009.

[12] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495, 2009.

[13] R. Koschke. Software Visualization in Software Maintenance, Reverse Engineering, and Re-engineering: A Research Survey. *Journal of Software Maintenance*, 15(2):87–109, 2003.

[14] D. Lawrie, C. Uehlinger, and D. Binkley. Vocabulary Normalization Improves IR-based Concept Location. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, pages 588–591, 2012.

[15] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, pages 18–18, 2005.

[16] J. I. Maletic, M. L. Collard, and A. Marcus. Source Code Files As Structured Documents. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 289–292, 2002.

[17] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2), 2004.

[18] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Folding Repeated Instructions for Improving Token-Based Code Clone Detection. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 64–73, 2012.

[19] D. Rattan, R. Bhatia, and M. Singh. Software Clone Detection: A Systematic Review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[20] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[21] C. K. Roy and J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

[22] M.-A. D. Storey, D. Čubranić, and D. M. German. On the Use of Visualization to Support Awareness of Human Activities in Software Development: A Survey and a Framework. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 193–202, 2005.

[23] F. Zhang and E. H. D'Hollander. Using Hammock Graphs to Structure Programs. *IEEE Transactions on Software Engineering*, 30(4):231–245, Apr. 2004.

[24] T. Zimmermann, R. Premraj, and A. Zeller. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, pages 9–15, 2007.

(注5)：https://github.com/YoshikiHigo/JCodeFlattener