特別研究報告

題目

凝集度メトリクス COB を用いた
類似メソッド集約範囲の決定支援手法

指導教員
井上 克郎 教授

報告者
井岡 正和

平成 23 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 22 年度 特別研究報告

凝集度メトリクス COB を用いた類似メソッド集約範囲の決定支援手法

井岡 正和

内容梗概

ソフトウェアの保守を困難にしている要因として，コードクローンが挙げられる．コードクローンとは，ソースコード中で互いに類似または一致したコード片のことである．コードクローンは，主にコピーアンドペーストや意図的に同一処理を繰り返し書くことによって生成される．もし，あるコードクローンを持つコード片に欠陥が含まれている場合，そのコードクローンすべてに対して修正を検討する必要がある．ソフトウェアの規模が非常に大きい場合は，このような保守作業には大きなコストがかかる．

保守コストを下げる方法の 1 つとして，コードクローンの集約が挙げられる．コードクローンの分類には，不一致部分を含むコードクローンがある．この分類に含まれるコードクローンは，あるコード片をコピーアンドペーストした後，文の挿入や削除を行うことで生成される．このようなコードクローンを持つメソッド対は類似メソッド対と呼ばれる．類似メソッド対の集約は，不一致部分の修正と除去を考慮する必要がある．これは，Template Method パターンに基づくリファクタリングパターンである Template Method の形成を行うことで達成できる．しかし，Template Method の形成は複数のリファクタリングパターンを利用するのでリファクタリングの熟練者でない限り難しい．

この類似メソッド対の集約を支援する手法を政井らが提案している．この手法は，不一致部分を含み，メソッドとして抽出することが可能なコード片の候補を提示することで，Template Method パターンの適用を支援している．

しかし，この手法は，対象とするメソッド対によっては候補数が 10 万を超えてしまうという問題がある．利用者が非常に多くの候補から有用な候補を見つけ出すのは現実的ではない．

そこで，本研究では，ソースコードブロック間の凝集度を用いて先ほどの候補の順位付けを行い，利用者にとって有用な候補を上位に提示することを支援する．凝集度とは，値が大きいほど機能的なまとまりを持つとされるもので，本研究では，凝集度メトリクス COB(Cohesion Of Blocks) を使用した．

実験では，メソッド間クローン率の高いオープンソースソフトウェアに対して本手法を適用し，利用者にとって有用な候補が上位に現れていることを確認できた．

主な用語

コードクローン
リファクタリング
Template Method の形成
凝集度
メトリクス COB(Cohesion Of Blocks)

目 次

1 まえがき **5**

2 背景 **7**

2.1 コードクローン 7
2.2 リファクタリング 7
2.3 デザインパターン 8
2.4 デザインパターンを利用するリファクタリング 9
2.4.1 Template Method の形成 9
2.4.2 Template Method の形成の例 9
2.5 リファクタリング支援手法 11
2.5.1 類似メソッド集約候補を挙げる FTMPATool 11
2.5.2 メソッド抽出を支援する凝集度メトリクス COB 14
2.6 既存研究の問題点 15

3 提案手法 **17**

3.1 [ステップ 1]FTMPATool が出力する集約候補の取得 17
3.2 [ステップ 2] メソッド抽出候補の大きさに基づく集約候補のフィルタリング . 19
3.3 [ステップ 3] メソッド抽出候補に対する凝集度の算出 22
3.4 [ステップ 4] 集約候補に対する凝集度の計算 22
3.5 [ステップ 5] 凝集度に基づく集約候補の順位付け 22

4 適用実験 **26**

4.1 準備 26
4.2 実験 26
4.3 考察 26

5 関連研究 **31**

5.1 Juillerat らの手法 31
5.2 兼光らの手法 31
5.3 Krinke の手法 32

6 まとめと今後の課題 **33**

謝辞 **34**

参考文献 **35**

付録 **37**

A 実験対象メソッド 37

1　まえがき

ソフトウェアの保守を困難にしている要因として，コードクローンが挙げられる．コードクローンとは，ソースコード中で互いに類似または一致したコード片のことである [9]．コードクローンは，主にコピーアンドペーストや意図的に同一処理を繰り返し書くことによって生成される．もし，あるコードクローンを持つコード片に欠陥が含まれている場合，そのコードクローンすべてに対して修正を検討する必要がある．ソフトウェアの規模が非常に大きい場合は，このような保守作業には大きなコストがかかる．つまり，コードクローンを効率的に集約し，取り除くことができれば，保守コストを下げることができる [15, 16, 20]．

コードクローンの分類には，不一致部分を含むコードクローンがある．この分類に含まれるコードクローンは，あるコード片をコピーアンドペーストした後，文の挿入や削除といった修正作業を行うことで生成される．このようなコードクローンを持つメソッド対は類似メソッド対と呼ばれる [17]．類似メソッド対の集約は，不一致部分の修正と除去を考慮する必要があるため，完全に一致するコードクローンの集約に比べて，困難である．

コードクローンを取り除く手法に，リファクタリングがある．リファクタリングとは，“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整理すること．” である [6]．このリファクタリングの手法として，Kerievsky はデザインパターンを使用する手法を提案している [13]．ソフトウェア開発におけるデザインパターンとは，過去の設計者が編み出した典型的な問題に対する解決策をカタログ化したものである．Kerievsky が文献 [13] の中で提案しているリファクタリングパターンの中に，コードクローンに対して効果的なリファクタリングを行う “Template Method の形成” がある．

“Template Method の形成” とは，GoF デザインパターンの 1 つである Template Method パターンに基づくリファクタリングパターンである [7]．Template Method パターンとは，親クラスでおおまかな処理のアルゴリズムを決めておき，具体的な内容を子クラスに任せるものである．そのため，子クラスでは共通の処理を実装する必要がなく，クラス間で類似したコード片の出現を避けることができる．

類似メソッド対の集約を支援する手法を政井らが提案している [17]．この手法は，不一致部分を含み，メソッドとして抽出することが可能なコード片の候補を提示することで，Template Method パターンの適用を支援している．しかし，対象とするメソッド対によっては提示される候補数が 10 万を超えてしまうという問題がある．利用者が非常に多くの候補から利用者にとって有用な候補を見つけ出すの現実的ではない．

そこで，本研究では，候補の順位付けを行い，利用者にとって有用な候補を上位に提示することを支援する．Template Method パターンでは，不一致部分を包括するようにメソッドとして抽出するので，このメソッドとして抽出する範囲に機能的なまとまりがあれば良い

と考えた．そこで，メトリクスとして，値が大きいほどソースコードブロック間で機能的なまとまりを持つとされる凝集度を使用し，降順で順位付けを行った．

凝集度として，計算量の少ない三宅らが提案したCOB(Cohesion Of Blocks) を用いる [18]．COB は，特定のデータ要素と協調する機能要素の割合の平均に着目しており，メソッドの構成要素の協調度合を示す．

メソッド間クローン率 [17] の高いオープンソースソフトウェアに対して本手法を適用し，評価実験を行った．利用者にとって有用な候補が上位に現れていることを確認できた．

以降，2 節では本研究に関連する用語を説明する．3 節では，候補の順位付けの手法について説明し，4 節では適用実験について述べる．そして，5 節で関連研究を述べ，6 節で本研究のまとめと今後の課題について述べる．

2　背景

本研究の提案手法の背景として，コードクローン，リファクタリング，デザインパターン，デザインパターンを利用するリファクタリング，リファクタリング支援手法，既存研究の問題点について説明する．

2.1　コードクローン

コードクローンとは，ソースコード中で互いに類似または一致したコード片のことである．もし，あるコードクローンを持つコード片に欠陥が含まれている場合，そのコードクローンすべてに対して修正を検討する必要がある．ソフトウェアの規模が非常に大きい場合は，このような保守作業には大きなコストがかかる．そのため，コードクローンを取り除くことは保守の観点で重要である．

発生要因

コードクローンの発生要因は主に以下のものが挙げられる [4].

コピーアンドペーストによる既存コードの再利用

正常に動作する既存コードをコピーアンドペーストし，一部を修正して再利用することが多い．

コーディングスタイル

エラー表示やインターフェース表示等の単純なコードはクローンになりやすい．

定型処理

繰り返し書く処理は，プログラマーが処理手順を覚えてしまうので，似たコードを書く傾向がある．

データ構造の違い

同じ処理であるが異なるデータ構造の場合に，ライブラリを使わず同じ処理を書くことが多い．

パフォーマンス改善

時間制約が厳しいシステムにおいて，コンパイラがインライン展開を提供していない場合に，意図的に繰り返し処理を記述して最適化することがある．

2.2　リファクタリング

リファクタリングとは，“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整理すること” である．Fowler は，設計の問題を解

消するためにどのようなリファクタリングが役に立つかを説明している [6]．設計の問題の中にコードクローン (文献 [6] では “Duplicated Code”) があり，このリファクタリングについて以下のパターンが挙げられている．

メソッド抽出

メソッド内からひとまとめにできるコード片を抽出し，新たなメソッドとして定義するリファクタリングパターンである．このとき，メソッドに意味のある名前をつけることが重要である．同じクラス内に，抽出したコード片のコードクローンがある場合は，コード片の代わりに抽出メソッドを呼び出すことでコードクローンを取り除くことができる．

メソッド引き上げ

複数の子クラスで定義された同じ結果をもたらすメソッドを共通の親クラスに引き上げるリファクタリングパターンである．このリファクタリングを行うことでコードクローンを取り除くことができる．メソッド内の一部のみがコードクローンである場合は，上記メソッドの抽出を行い，抽出したメソッドを引き上げることでコードクローンを取り除くことができる．

2.3 デザインパターン

ソフトウェア開発におけるデザインパターンとは，過去の設計者が編み出した典型的な問題に対する解決策をカタログ化したものである．一般に有名なデザインパターンに GoF デザインパターンがある [7]．GoF デザインパターンの例として，後述の FTMPATool で利用されている Template Method パターンについて説明する．

Template Method パターン

Template Method パターンとは，アルゴリズムの骨格を親クラスで実装し，具体的な実装は子クラスに任せるデザインパターンである [7].

親クラスのメソッドで処理の順序は実装するが，そのメソッド内で呼び出されている一部のメソッドは同じクラス内で抽象メソッドとして定義する．親クラスで定義された抽象メソッドをそれぞれの子クラスでオーバーライドすることによって，子クラスの目的に応じた処理を実装することができる．つまり，Template Method パターンを用いるよって，共通の処理を各子クラスで実装する必要がなくなる．また，似た処理を行う子クラスを作成する場合もメソッドをオーバーライドするだけで良いので，拡張が容易である．しかし，Template Method パターンは注意して使用しないとソフトウェアが複雑になる恐れがある [12].

2.4 デザインパターンを利用するリファクタリング

Kerievsky は，デザインパターンを利用するリファクタリングについて，次のように述べている [13]．“デザインパターンの構造を目指したコーディングは，あくまでデザインパターンの構造を作るだけであり，設計者のニーズに答えているとはいえない．必要に応じて，状況に適したデザインパターンを選び，設計をリファクタリングすることが望ましい．”

以下で，コードクローンを取り除くためのデザインパターンを利用するリファクタリングパターンである，Template Method の形成について説明する．

2.4.1 Template Method の形成

Template Method の形成とは，親クラスが共通で，類似メソッドを持つ子クラス間に，前節 2.3 で述べた Template Method パターンを適用するリファクタリングパターンである [6, 13]．

Template Method の形成は，以下の手順で行う．

1. 類似メソッド間の不一致部分を見つける．
2. ステップ 1 で見つけた不一致部分を含むように，子クラスにメソッドとして抽出するコード片を決定する．
3. ステップ 2 で決定したコード片を子クラスにメソッドとして抽出し，元のコード片をそのメソッドの呼び出し文に置き換える．
4. 記述を揃えたメソッドを親クラスに引き上げる．また，親クラスにステップ 3 で抽出したメソッドを抽象メソッドとして定義する．

2.4.2 Template Method の形成の例

図 1 を用いて説明する．図 1 の上側が Template Method の形成前，下側が形成後となっている．また，ResidentialSite クラスの getBillableAmount() メソッドと LifelineSite の getBillableAmount() メソッドが類似メソッド対である．

1. 図 1 の類似メソッド対は，戻り値となる式は同じであるが，各変数の宣言文が異なっており，差分となっている．
2. ステップ 1 の差分を含むように，base，tax の宣言文を抽出するコード片とする．

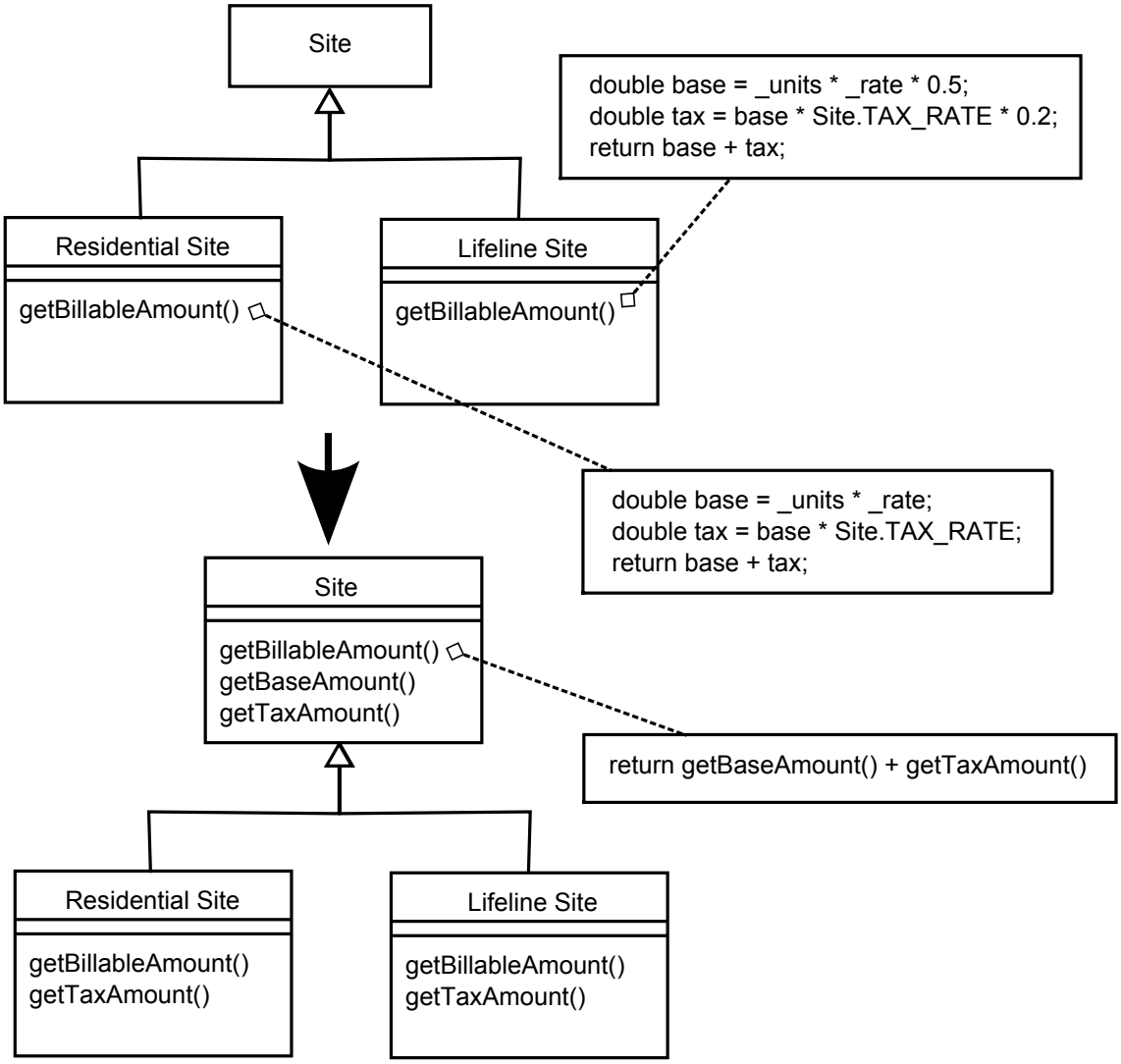


図 1: Template Method の形成の例 [6]

3. ステップ2で決定したコード片をResidentialSiteクラス，LifelineSiteクラスそれぞれに対して，baseの宣言文をgetBaseAmount()メソッド，taxの宣言文をgetTaxAmount()メソッドとして抽出する．また，元のコード片を抽出したメソッドに置き換える．

4. 記述を揃えたgetBillableAmount()メソッドを親クラスであるSiteクラスに引き上げる．また，ステップ3で抽出したメソッドであるgetBaseAmount()とgetTaxAmount()をSiteクラスに抽象メソッドとして宣言する．

2.5 リファクタリング支援手法

多くのリファクタリングを支援する手法がある．ここでは，Template Methodの形成を用いた類似メソッド集約候補を挙げるFTMPAToolと，メソッド抽出の必要性を評価するための凝集度メトリクスCOBについて説明する．

2.5.1 類似メソッド集約候補を挙げるFTMPATool

FTMPAToolは政井らが提案した手法を実装したツールで，Template Methodの形成を用いた類似メソッド集約の支援を行う[17]．このツールは，統合開発環境Eclipse[5]のJava開発キットのプラグインとして実装されているので，Eclipseの既存の機能を利用でき，また，Eclipseを用いたコーディングの過程で利用できる．

まず，この手法で利用している類似文字列マッチングアルゴリズムを説明し，その後，この手法の手順を説明する．入力として類似メソッド対が与えられるとする．

類似文字列マッチングアルゴリズム

類似文字列マッチングアルゴリズム[19]は，動的計画法を用いて，与えられた2つの文字列がどれだけ類似しているかを判定するアルゴリズムである．このアルゴリズムでは，2つの文字列より図2のような表を作成する．表の1行目には0を，1列目には0から順に列の最後まで数字を格納する．他のセルには，各行，列に対応する2つの文字列を次の方法で比較した結果を格納する．

同じ文字の場合 左上のセルの値を格納する．

異なる文字の場合 左，上，左上のセルが格納している値のうち，最小の値に1足した値を格納する．

探索は，以下の手順で行う(図3)．

1. 最終行，最終列のセルから探索を開始する．

		a	d	e	f	c
	0	0	0	0	0	0
a	1	0	1	1	1	1
b	2	1	1	2	2	2
c	3	2	2	2	3	2

図 2: 文字列比較表

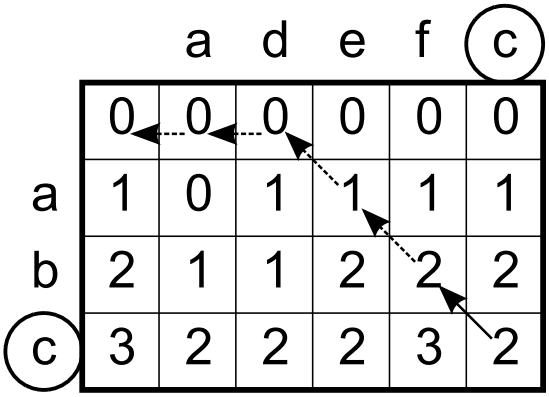


図 3: 文字列比較表の探索

2. 左，上，左上のセルの中で，格納している値が最小のセルに移動する．複数のセルが格納している値が最小値の場合は，左上，左，上の順に優先して移動する．
3. 1 行目，1 列目に到達したら探索を終了する．

表の探索において，左上に移動し，かつ，値が変化しなければ文字が一致している．

[ステップ 1] 抽象構文木の生成

Eclipse の抽象構文木生成機能を用いて，与えられた類似メソッド対の抽象構文木をそれぞれ生成する．この抽象構文木のノードは，大きく以下の 2 つに分類される．

タイプ A(値や子ノードを持つノード)

“ユーザ定義名”，“return 文”，“代入文” 等のノード.

タイプ B(子ノードの列を持つノード)

中括弧で括られた範囲に対応するノード.

[ステップ 2] 差分となる部分木の検出

ステップ 1 で生成された 2 つの抽象構文木を比較し，抽象構文木間の差分となる部分木を検出する．抽象構文木の比較は，メソッド宣言に対応するノードから開始し，子ノードへと再起的に比較を繰り返すことで行う．ノードの比較は，主に以下の操作からなる．

種類の比較

ノードの種類が同じか比較する．異なっていれば差分として検出し，同じであればタイプによって以下の比較を行う．

値，子ノードの比較 (タイプ A)

タイプ A のノードの場合，ノードの持つ値や子ノードを比較する．異なっていれば差分として検出する．

列の比較 (タイプ B)

タイプ B のノードの場合，子ノードの列を比較する．比較には，先ほど説明した類似文字列マッチングアルゴリズムを用いている．

差分はソースコード上においては，ステートメント単位である．また，差分と判断されたノードの子ノードはすべて差分と判断される．

[ステップ 3] 抽出が容易な部分木の検出

差分となる部分木を含む，メソッドとして抽出可能な部分木を検出する．

メソッドとして抽出が可能か否かは，Eclipse のリファクタリング機能である “メソッド抽出” を行うための事前条件判定機能を用いて判定している．この事前条件が抽出不可と判定するもので，この手法に関係するものは以下の 3 つである．

条件 1 複数の変数の初期化を含む宣言文，または，複数の変数への代入文が含まれており，かつ，それらの変数が後のコードにおいて参照されている．

条件 2 break 文，continue 文が含まれているが，これらに対応する制御文が含まれていない．

条件 3 戻り値を持たない return 文を含んでいる．

これらの条件を満たすコード片は，そのままメソッドとして抽出することができない，または，抽出した場合に動作の保証ができない．よって，このようなコード片はメソッドとして抽出不可とし，段階的に範囲を拡大して条件判定を繰り返すことで抽出可能であるコード片を表す部分木列の検出を行う．

また，抽出範囲すべての組み合わせが検出されるように，差分となる部分木や抽出可能として検出された部分木からも範囲の拡大を行う．

[ステップ 4] 部分木列の分類

メソッド抽出可能であると検出された部分木列が表すコード片を，実際にメソッドとして抽出した後のメソッド呼び出し文の差異に基づいて分類する．以下の条件を用いて，6 つに分類する．

条件 1 戻り値の型，または値を戻す変数が異なる．

条件 2 引数として渡す変数が異なる．

条件 3 類似メソッド対の片方にのみ，対応する位置にコード片がない．

分類 1 条件 1，条件 2 を満たす抽出箇所が存在しない．

分類 2 条件 1 を満たす抽出箇所が 1 つ存在する．

分類 3 条件 2 を満たす抽出箇所が 1 つ存在する．

分類 4 条件 1，および条件 2 を満たす抽出箇所が 1 つ存在する．

分類 5 条件 3 を満たす抽出箇所が 1 つ存在する．

分類 6 条件 1~3 を 1 つ以上満たす抽出箇所が複数存在する．

分類 1 はそのままメソッドの抽出を行うことができるため，Template Method の形成が容易である．その他の分類は抽出を行った後の記述が揃わないため，事前にソースコードの修正が必要であり，Template Method の形成が容易とはいえない．

2.5.2 メソッド抽出を支援する凝集度メトリクス COB

COB(Cohesion Of Blocks) は三宅らが提案したメトリクスで，メソッド抽出の必要性を評価する凝集度メトリクスである [18]．凝集度とは，モジュール内の構成要素が特定の機能を実現するために協調している度合いを表す．

メソッドの構成要素の協調度合いを表すために，メソッド内で使用されている変数をデータ要素，コードブロックを機能要素とみなすと，メトリクス COB は式 (1) で定義される．b はメソッド内のコードブロック数，v はメソッド内で使用されている変数の数，V_j はメソッド内で使用されている j 番目の変数，$\mu(V_j)$ は変数 V_j を使用しているコードブロック数を示す．

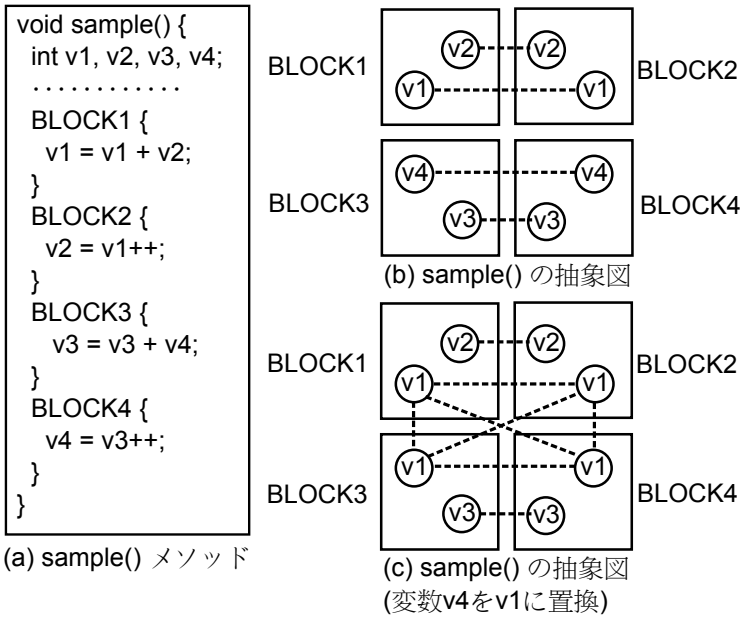


図 4: サンプルコード [18]

$$COB = \frac{1}{b}\frac{1}{v}\sum_{j}^{v}\mu(V_j) \quad (0 \leq COB \leq 1) \tag{1}$$

図 4(a) の sample() メソッドのコードを用いた例を挙げる．図中の BLOCK1∼4 はコードブロックを表す．図 4(b) は，sample() メソッドのコードブロックと変数の協調関係の抽象図である．四角はコードブロック，四角の中の丸はコードブロック中で使用されている変数，点線はコードブロック間で変数が協調していることを表す．図 4(b) より，sample() メソッドは BLOCK1 と BLOCK2 は変数 v1，v2 を介して協調しており，同様に，BLOCK3 と BLOCK4 は変数 v3，v4 を介して協調している．一方，BLOCK1，2 と BLOCK3，4 は協調していない．このときの COB の値は 0.5 となる．また，図 4(c) は，(b) での変数 v4 を v1 に置き換えたものである．このようにしたとき，すべてのコードブロックが変数 v1 を介して協調するため，COB の値は大きくなり，値は 0.66 になる．

このことから，メソッド内のコードブロック間で変数が互いに協調しているときに COB の値が大きくなることが分かる．つまり，メソッド内のコードブロック間で変数が互いに協調していない，すなわち，COB の値が小さいときはメソッドを分割すべきであるといえる．

2.6 既存研究の問題点

政井らの提案した手法は主に 2 つの問題がある．

1. 候補を提示する際に，利用者にとって有用な候補から順に表示されない．

2. 入力として与えるメソッド対によっては，候補が膨大な数になる．

以下で，各問題の原因について説明する．

1．提示される候補の順序に意味を持たない

候補は各差分となるコード片から範囲を段階的に拡大させて検出する．その後，この候補群を抽出時のメソッド呼び出し文の差異に基づいて分類しているが，候補自体のソーティングは行っていない．そのため，提示される候補の順序に意味を持たないので，利用者にとって有用な候補を見つけ出すのは難しい．

2．すべての組み合わせの候補を提示する

FTMPATool では，抽出範囲すべての組み合わせが検出されるように候補を挙げる．そのため，類似メソッド対両方の行数が大きいと候補数が膨れ上がってしまう．候補数が 10 万を超えることもあり，すべてを確認することは現実的ではない．

3　提案手法

本研究では，メトリクス COB を用いて Template Method パターン適用時の抽出メソッドに機能的なまとまりのあるものから利用者に提示することで，前節 2.6 で述べた問題点の解決をはかる．機能的なまとまりを持つものが利用者にとって有用であると考えたので，利用者に機能的なまとまりのあるものから提示することで問題点の 1 つ目を解決できる．また，機能的なまとまりのないものを候補に挙げないことで問題点の 2 つ目を解決できる．

入力として類似メソッド対が与えられるものとし，処理は以下の 5 ステップで行う．図 5 は提案手法全体の流れを示す.

ステップ 1 FTMPATool が出力する集約候補の取得

ステップ 2 メソッド抽出候補の大きさに基づく集約候補のフィルタリング

ステップ 3 メソッド抽出候補に対する凝集度の算出

ステップ 4 集約候補に対する凝集度の計算

ステップ 5 凝集度に基づく集約候補の順位付け

FTMPATool では不一致部分を検出する際に文献 [19] の類似文字列マッチングアルゴリズムをノード単位で適用して用いている．しかし，現在適用されている方法では一致している箇所を不一致と判定することがある．そこで，この問題を次の方法で改善した.

不一致部分検出アルゴリズムの改善

類似文字列マッチングアルゴリズムを前節 2.5.1 で説明したが，図 6 のように，“abc”，“adefc” という同じ文字列対でも行と列の長さに依存して結果が異なる.

図 6 より，行のほうの文字列が短い場合に，先頭の文字が差分として検出されてしまう．したがって，行のほうの文字列が長いほうが良いと考え，不一致部分検出アルゴリズムに適用した.

3.1　[ステップ 1]FTMPATool が出力する集約候補の取得

FTMPATool 実行し，各候補の抽出コード片情報を取得する．FTMPATool からテキストファイルとして，メソッド対のファイルパス，メソッドの位置，各候補ごとの抽出コード片の位置が出力される．具体例を同じ親クラスを持つ 2 つの子クラスの draw() メソッドを用いて挙げる．図 7 に類似メソッド対と候補の例を示す.

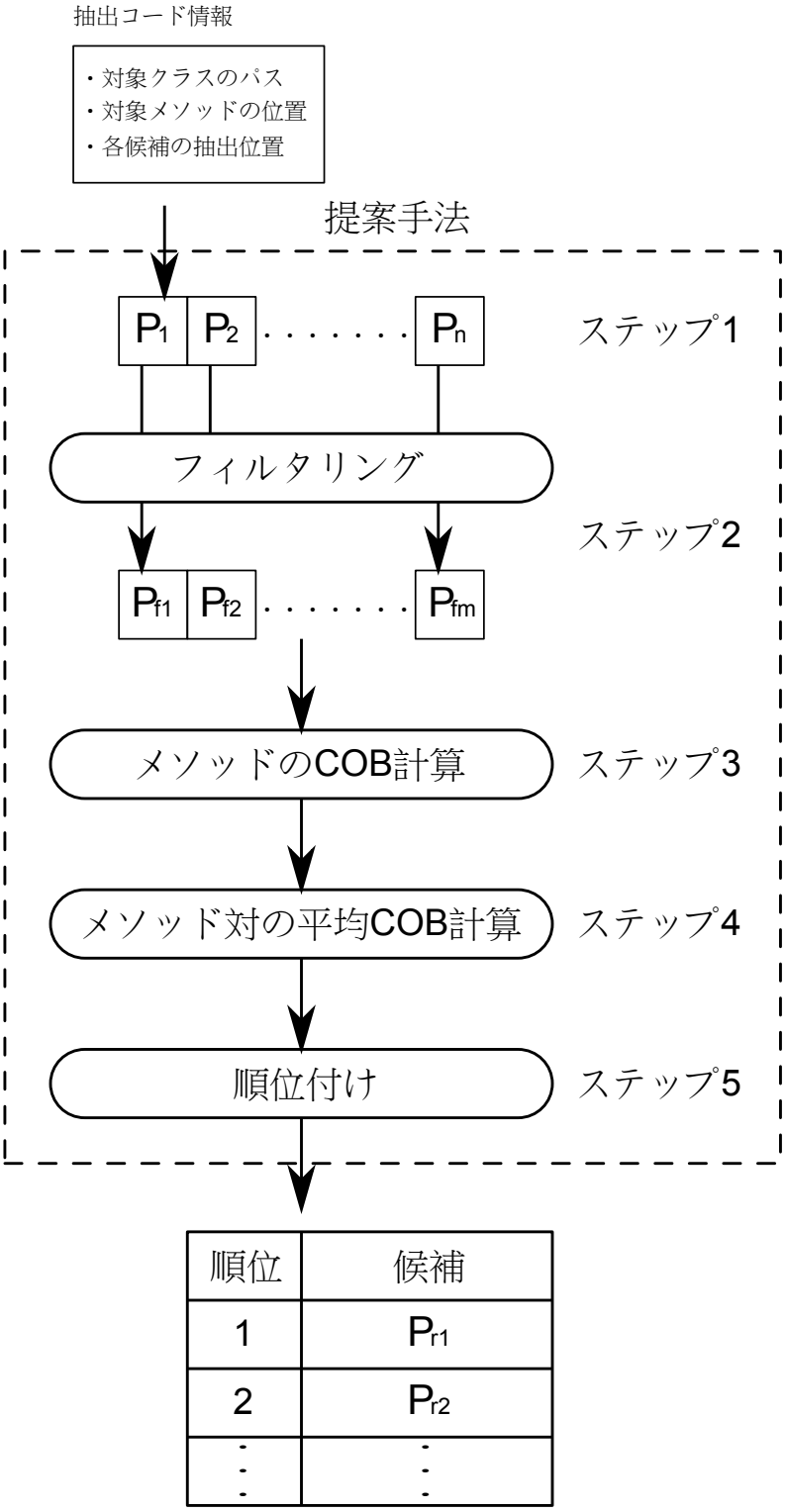


図 5: 提案手法全体の流れ

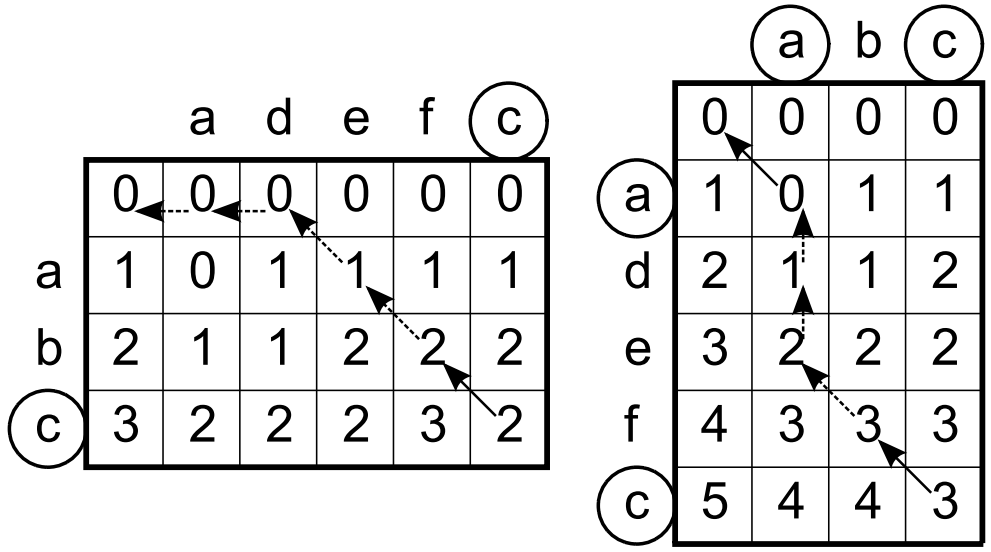


図 6: 行，列を入れ替えた比較結果

この出力ファイルより 2 つのメソッドを取得し，COB の計算に不必要なものを削除するための前処理を行う．前処理の内容は以下の通りである．図 9 に前処理の例を示す．以降では，見やすさと紙面の都合上，前処理をしていない状態の図で説明する．

- コメントの削除
- 文字列の削除
- 型パラメータの削除
- 空白の削除
- if 文等で中括弧の補完 (図 8)
- 代入のない宣言の削除

3.2 [ステップ 2] メソッド抽出候補の大きさに基づく集約候補のフィルタリング

抽出元メソッドのコードの大きさに対して閾値を設定し，抽出するコード片の大きさがその閾値を超えるコード片が 1 つでもある場合は，フィルタリングを行い，候補に挙げない．フィルタリングの例を図 10 に示す．候補 1，2 は，各抽出コード片のステートメント数がメソッド全体のステートメント数の半分以下であるので，使用する候補として分類した．一方，候補 3 は，抽出コード片のステートメント数がメソッド全体のステートメント数の 9 割弱にも及ぶので，除外される候補として分類した．

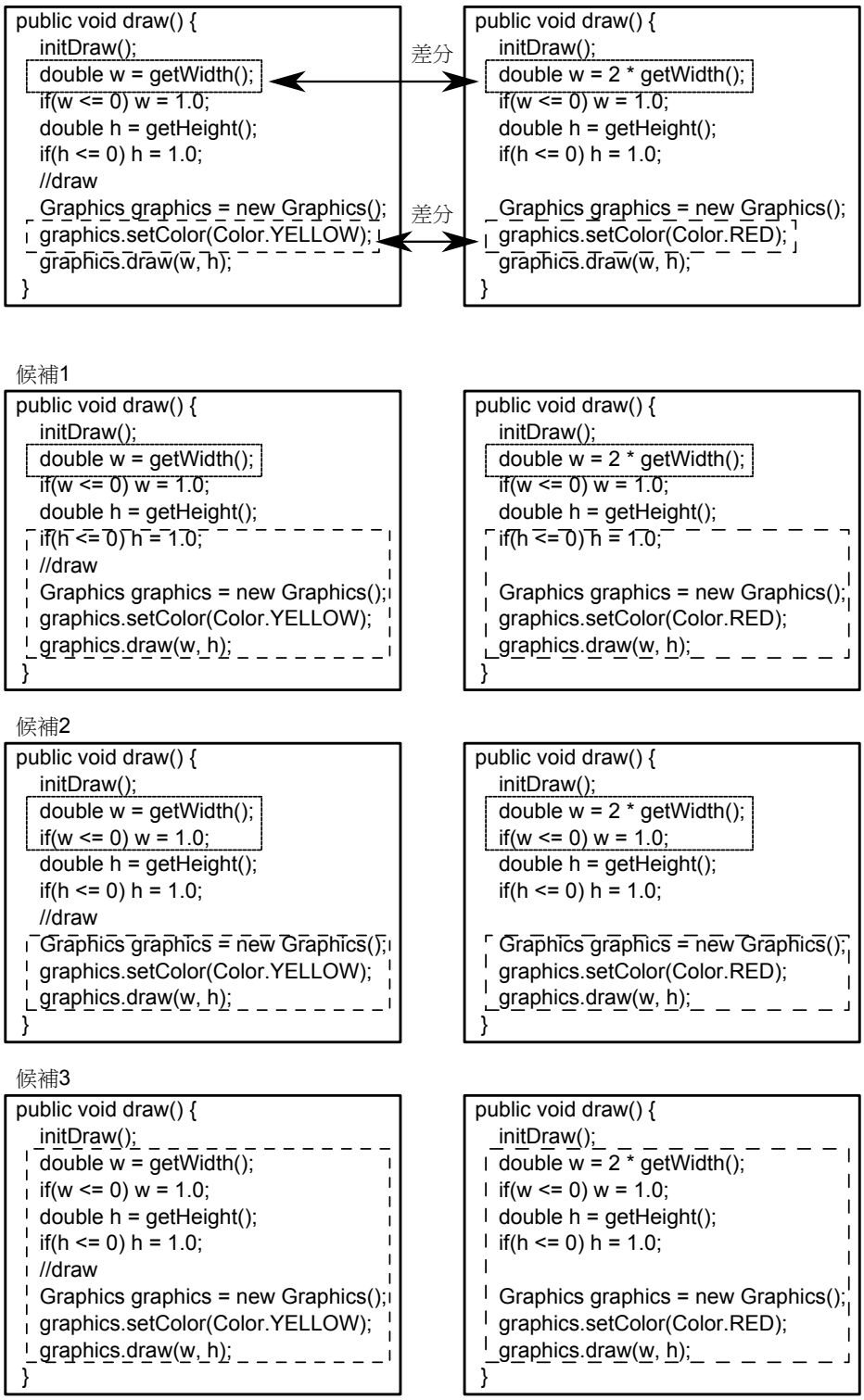

public void draw() {
initDraw();
double w = getWidth();
if(w <= 0) w = 1.0;
double h = getHeight();
if(h <= 0) h = 1.0;
//draw
Graphics graphics = new Graphics();
graphics.setColor(Color.YELLOW);
graphics.draw(w, h);
}
差分
差分
public void draw() {
initDraw();
double w = 2 * getWidth();
if(w <= 0) w = 1.0;
double h = getHeight();
if(h <= 0) h = 1.0;
Graphics graphics = new Graphics();
graphics.setColor(Color.RED);
graphics.draw(w, h);
}
候補1
候補2
候補3

図 7: [ステップ 1] 候補例

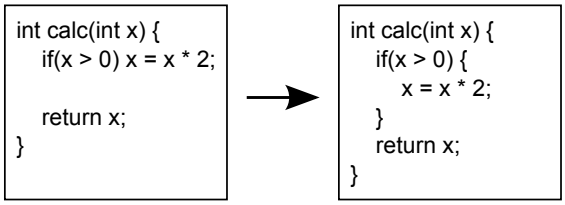


図 8: 中括弧の補完

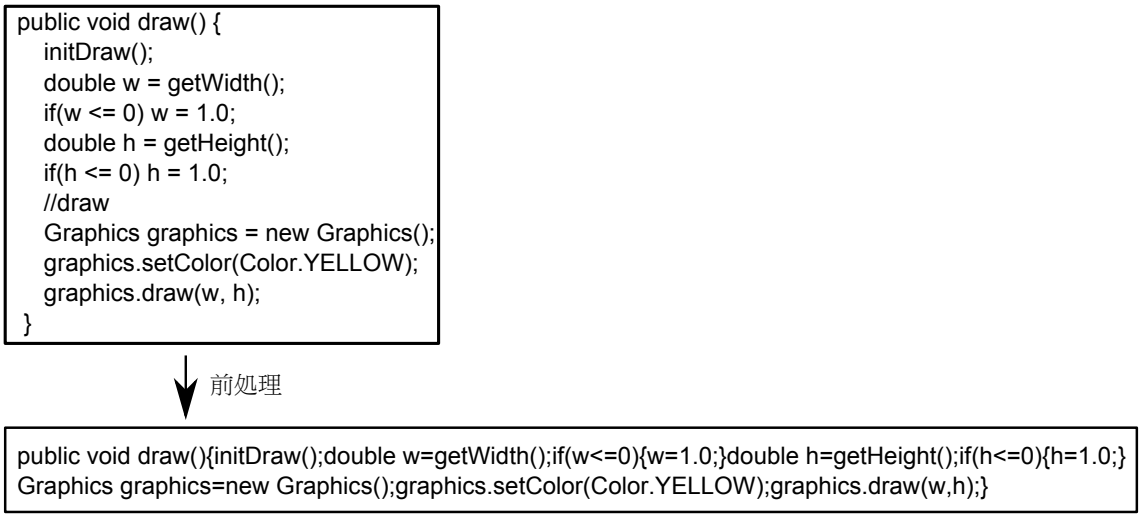


図 9: [ステップ 1] 前処理 (実際は前処理後のコードに改行は含まれない)

このフィルタリングを行うのは，大きい範囲をメソッドとして抽出した場合に，抽出したメソッド対が再びコードクローンとなるためである．

3.3 [ステップ 3] メソッド抽出候補に対する凝集度の算出

各抽出コード片を 1 つのメソッドと考えて COB を計算する．図 11 のように，中括弧で括られている箇所を 1 つのブロックとする．なお，中括弧が入れ子になっている場合は，中括弧で括られている一番外側を 1 つのブロックとし，内側はブロックとみなさない．また，ブロックとブロックの間のコード片を 1 つのブロックとみなす．図 12 に COB 計算の例を示す．候補 1 の左側のメソッドについて詳しく説明する．実線で括られている抽出コード片では，ブロックが 1 つ，変数が 1 つであるので $COB = 1.0$ となる．破線で括られている抽出コード片では，if 文の条件式 (ブロック 1)，if 文内のブロック (ブロック 2)，その他のコード片 (ブロック 3) の 3 つのブロックに分けられ，変数は w，h，graphics，Color の 4 つである．また，変数 w はブロック 3 で，変数 h はブロック 1，2，3 で，変数 graphics はブロック 3 で，変数 Color はブロック 3 で使用されている．よって，$COB = \frac{1}{3}\frac{1}{4}(3+1+1+1) = \frac{1}{2}$ となる．

3.4 [ステップ 4] 集約候補に対する凝集度の計算

ステップ 3 で求めた各抽出コードへの COB の値を用いて，類似メソッド対に含まれるすべての抽出コード片の COB の値の平均を計算する (図 12)．

3.5 [ステップ 5] 凝集度に基づく集約候補の順位付け

FTMPATool における 6 つの分類ごとに，類似メソッド対の平均 COB 値の大きさ順に順位付けを行う．このとき，類似メソッド対内のすべての抽出コード片がブロック 1 つのみを抽出する場合は，順位を低くする．これは，ブロック 1 つのみを抽出する場合は，必ず COB の値が 1 となるが，機能的なまとまりであるとはいえないことが多いためである．図 13 に順位付けした結果を示す．

使用する候補

候補1

```
public void draw() {
  initDraw();
  double w = getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;
  //draw
  Graphics graphics = new Graphics();
  graphics.setColor(Color.YELLOW);
  graphics.draw(w, h);
}
```

```
public void draw() {
  initDraw();
  double w = 2 * getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;

  Graphics graphics = new Graphics();
  graphics.setColor(Color.RED);
  graphics.draw(w, h);
}
```

候補2

```
public void draw() {
  initDraw();
  double w = getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;
  //draw
  Graphics graphics = new Graphics();
  graphics.setColor(Color.YELLOW);
  graphics.draw(w, h);
}
```

```
public void draw() {
  initDraw();
  double w = 2 * getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;

  Graphics graphics = new Graphics();
  graphics.setColor(Color.RED);
  graphics.draw(w, h);
}
```

除外される候補

候補3

```
public void draw() {
  initDraw();
  double w = getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;
  //draw
  Graphics graphics = new Graphics();
  graphics.setColor(Color.YELLOW);
  graphics.draw(w, h);
}
```

```
public void draw() {
  initDraw();
  double w = 2 * getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;

  Graphics graphics = new Graphics();
  graphics.setColor(Color.RED);
  graphics.draw(w, h);
}
```

図 10: [ステップ 2] フィルタリング例

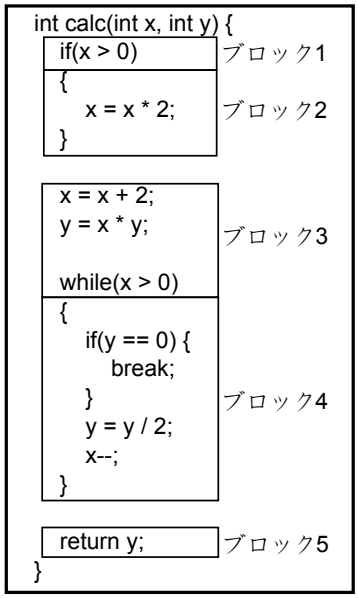


図 11: ブロック分けの例

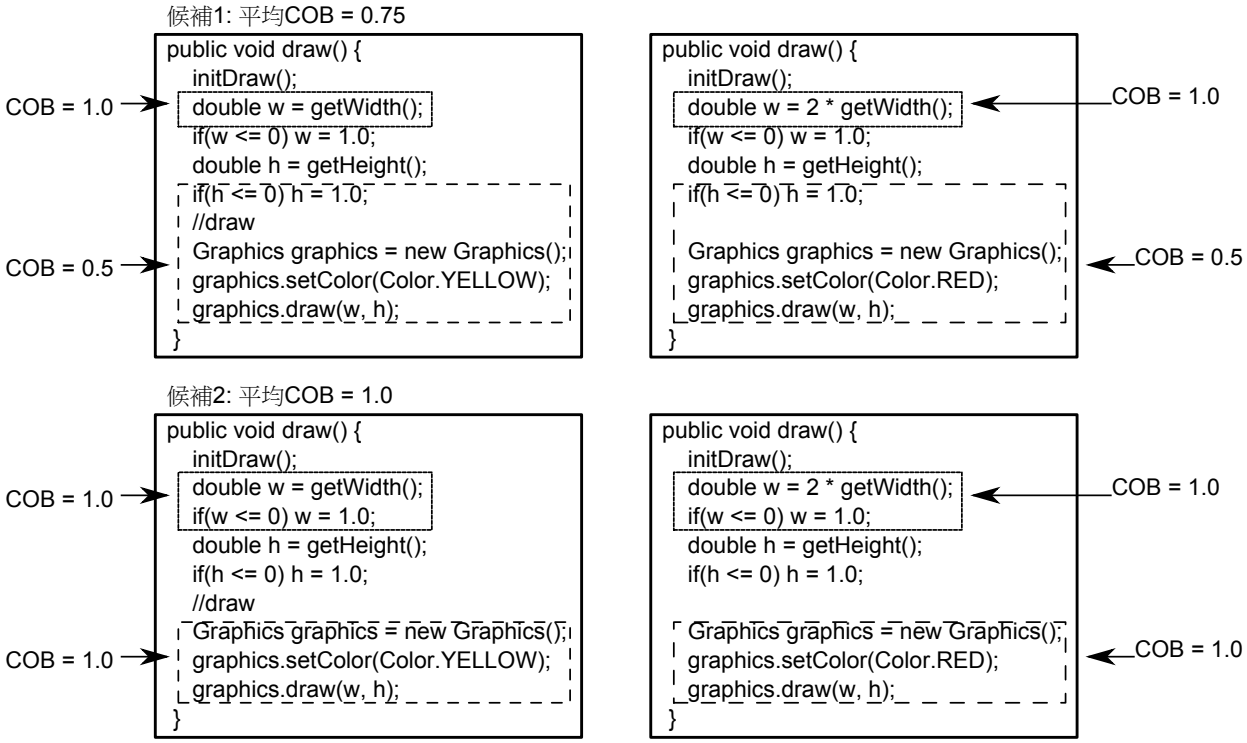


図 12: [ステップ 3，4]COB の計算例

1位 候補2: 平均COB = 1.0

```
public void draw() {
  initDraw();
  double w = getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;
  //draw
  Graphics graphics = new Graphics();
  graphics.setColor(Color.YELLOW);
  graphics.draw(w, h);
}
```

```
public void draw() {
  initDraw();
  double w = 2 * getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;

  Graphics graphics = new Graphics();
  graphics.setColor(Color.RED);
  graphics.draw(w, h);
}
```

2位 候補1: 平均COB = 0.75

```
public void draw() {
  initDraw();
  double w = getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;
  //draw
  Graphics graphics = new Graphics();
  graphics.setColor(Color.YELLOW);
  graphics.draw(w, h);
}
```

```
public void draw() {
  initDraw();
  double w = 2 * getWidth();
  if(w <= 0) w = 1.0;
  double h = getHeight();
  if(h <= 0) h = 1.0;

  Graphics graphics = new Graphics();
  graphics.setColor(Color.RED);
  graphics.draw(w, h);
}
```

図 13: [ステップ 5] 順位付け例

4　適用実験

3 節で述べた提案手法を，実際のソースコードを用いて適用実験を行った．

提案手法では，FTMPATool の問題点を改善することを目的としている．そのため，適用実験では，FTMPATool の出力と本手法を用いた場合の結果を比較する．

4.1　準備

適用実験の対象には，Antlr[2]，Apache Ant[1]，Azureus[3] の 3 つのオープンソースソフトウェアを用いた．また，提案手法を適用する類似メソッド対を見つけるために，コードクローン検出ツールとして Scorpio[8] を用いた．Scorpio は，プログラム依存グラフを用いたコードクローン検出ツールであり，不一致部分を含むコードクローン，すなわち，類似メソッド対を検出できる．そして，Scorpio で検出した結果を用いてメソッド間クローン率 [17] を計算し，その値が大きいメソッド対を実験の対象とした．なお，Template Method の形成が容易である分類 1 の候補のみを対象に実験を行う．各実験対象の FTMPATool の出力における分類 1 の候補数は，Antlr プロジェクトは 34，Ant プロジェクトは 23，Azureus プロジェクトは 479 である．

利用者に優れた候補を提示できていることを確認するために，実験として，FTMPATool の出力の表示順の上位 10 件と本手法の順位付けの結果の上位 10 件の中で，どの候補が優れた候補であるかを判断してもらう．

被験者は，コンピュータサイエンスを専攻している学部 4 年から博士前期課程 2 年の学生 9 人を対象にした．

4.2　実験

Antlr プロジェクトの CppCodeGenerator クラスの genErrorHandler() メソッドと JavaCodeGenerator クラスの genErrorHandler() メソッド，Ant プロジェクトの Arc クラスの executeDrawOperation() メソッドと Ellipse クラスの executeDrawOperation() メソッド，Azureus プロジェクトの MD5 クラスの digest() メソッドと SHA1 クラスの digest() メソッドの 3 つの類似メソッド対に対して評価実験を行ったところ，表 1，2，3 の結果が得られた．ただし，Ant プロジェクトの対象については，フィルタリングの結果候補数が 6 件となった．各メソッドのソースコードを付録に掲載する．

4.3　考察

全体の候補数に対する優れた候補に選ばれた割合を図 14 に，半数以上の被験者が優れた候補と判断した候補の数の推移を図 15 に示す．

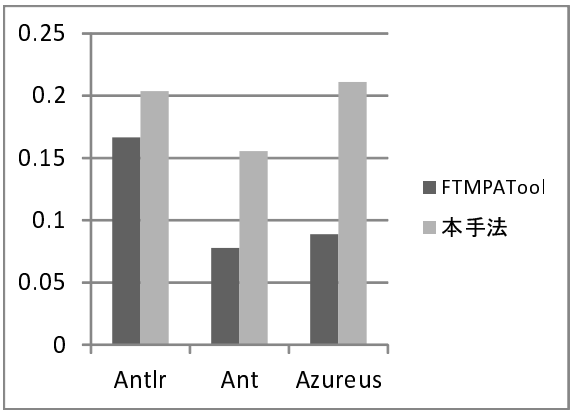


図 14: 優れた候補の割合

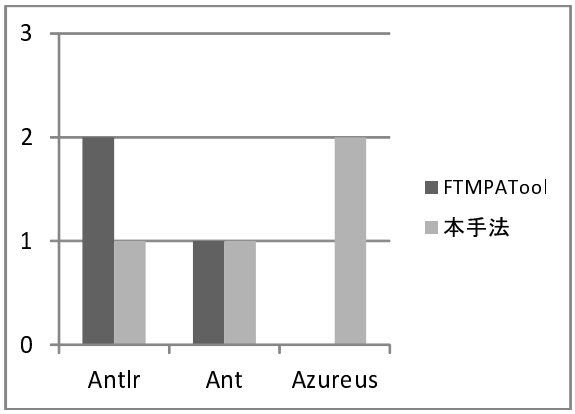


図 15: 半数以上が優れた候補と判断した数

表 1: Antlr 実験結果

(a) FTMPATool の結果

	候補 1	候補 2	候補 3	候補 4	候補 5	候補 6	候補 7	候補 8	候補 9	候補 10	候補選択率
被験者 A		○			○			○	○		0.40
被験者 B									○		0.10
被験者 C					○				○		0.20
被験者 D											0.00
被験者 E											0.00
被験者 F					○						0.10
被験者 G					○				○		0.20
被験者 H				○					○		0.20
被験者 I		○			○				○		0.30
選択率	0.00	0.22	0.00	0.11	0.56	0.00	0.00	0.11	0.67	0.00	

(b) 本手法の結果

	候補 1	候補 2	候補 3	候補 4	候補 5	候補 6	候補選択率
被験者 A							0.00
被験者 B							0.00
被験者 C						○	0.17
被験者 D							0.00
被験者 E	○	○			○	○	0.67
被験者 F		○				○	0.33
被験者 G	○		○			○	0.50
被験者 H							0.00
被験者 I						○	0.17
選択率	0.22	0.22	0.11	0.00	0.11	0.56	

表 2: Apache Ant 実験結果

(a) FTMPATool の結果

	候補 1	候補 2	候補 3	候補 4	候補 5	候補 6	候補 7	候補 8	候補 9	候補 10	候補選択率
被験者 A											0.00
被験者 B		○									0.10
被験者 C		○									0.10
被験者 D		○	○								0.20
被験者 E		○									0.10
被験者 F											0.00
被験者 G							○				0.10
被験者 H											0.00
被験者 I		○									0.10
選択率	0.00	0.56	0.11	0.00	0.00	0.00	0.11	0.00	0.00	0.00	

(b) 本手法の結果

	候補 1	候補 2	候補 3	候補 4	候補 5	候補 6	候補 7	候補 8	候補 9	候補 10	候補選択率
被験者 A					○				○		0.20
被験者 B						○					0.10
被験者 C						○	○				0.20
被験者 D						○	○				0.20
被験者 E					○	○					0.20
被験者 F						○					0.10
被験者 G	○					○	○				0.30
被験者 H											0.00
被験者 I							○				0.10
選択率	0.11	0.00	0.00	0.00	0.22	0.67	0.44	0.00	0.11	0.00	

表 3: Azureus 実験結果

(a) FTMPATool の結果

	候補 1	候補 2	候補 3	候補 4	候補 5	候補 6	候補 7	候補 8	候補 9	候補 10	候補選択率
被験者 A	○		○								0.20
被験者 B			○								0.10
被験者 C			○	○							0.20
被験者 D			○								0.10
被験者 E	○										0.10
被験者 F											0.00
被験者 G											0.00
被験者 H				○							0.10
被験者 I											0.00
選択率	0.22	0.00	0.44	0.22	0.00	0.00	0.00	0.00	0.00	0.00	

(b) 本手法の結果

	候補 1	候補 2	候補 3	候補 4	候補 5	候補 6	候補 7	候補 8	候補 9	候補 10	候補選択率
被験者 A								○		○	0.20
被験者 B										○	0.10
被験者 C							○			○	0.20
被験者 D							○	○	○	○	0.40
被験者 E							○		○	○	0.30
被験者 F							○				0.10
被験者 G			○						○		0.20
被験者 H							○		○		0.20
被験者 I								○		○	0.20
選択率	0.00	0.00	0.11	0.00	0.00	0.00	0.56	0.33	0.44	0.67	

図 15 より，どのプロジェクトについても，本手法を用いた場合に少なくとも 1 つは半数以上の被験者が優れた候補と判断した候補が存在する．また，図 14 より，どのプロジェクトについても，FTMPATool の結果に比べて本手法のほうが優れた候補が多いという結果となった．このことより，本手法を用いることによって，利用者にとって有用な候補を上位に提示することができている．つまり，凝集度による順位付けは妥当であるといえる．また，候補数が多い場合にも，候補の上位のみを提示すれば優れた候補を提示できるので，候補数が 10 万を超えるという問題を解消することができる．

表 1 より，Antlr プロジェクトについては，抽出範囲によるフィルタリングによって優れた候補を除外しているので，フィルタリングの閾値を対象によって柔軟に設定する必要があるといえる．フィルタリングの閾値を決定するために，閾値ごとの本手法の結果を用いた追加実験が必要である．

5　関連研究

5.1　Juillerat らの手法

Juillerat らは，Template Method の形成を自動的に行う手法を提案している [10]．Juillerat らの手法は，政井らの手法と同様に抽象構文木から差分を検出する．Juillerat らの手法では，差分を検出するために，2 つの抽象構文木を深さ優先探索の帰りがけ順に探索することで作成したノード列を用いて比較を行う．このことより，高速に比較を行うことができるが，抽象構文木の構造的な情報を失う．また，抽出範囲において，片方に足りない代入文がある場合は，対応する位置に同じ変数に値が変化しない代入文を追加し，抽出範囲を変化させない．これらによって，抽出を自動化することは可能であるが機能的なまとまりを抽出しているとはいえない．一方，本手法は，自動的に抽出することはできないが，凝集度を用いて機能的なまとまりを持ったものから利用者に提示しているため，利用者が適切であるものを選ぶことができる．

5.2　兼光らの手法

兼光らは，メソッド抽出リファクタリングの候補を挙げる手法を提案している [11]．本手法では，ブロック間の協調関係のみを見て機能的なまとまりかどうか判断しているが，兼光らの手法は，プログラム依存グラフを用いて機能的なまとまりかどうかを判断している．機能的なまとまりを判断する条件は以下の 3 つである．

ある変数が一度しか参照されない

データ依存片が 1 つのみ存在する場合であり，そのデータ依存片の両端のノードは同じメソッドにまとめておくべきである．

ある変数が多くの文で参照される

よく参照される変数は重要な値であり，その参照関係はつながりが強いといえる．

多くの変数がある文で参照される

複数の引数を取るメソッド呼び出し準備時に出現しやすく，まとまった機能を表しており，つながりが強いといえる．

兼光らは手法をツールとして実装しており，グラフによる可視化が直観的で抽出する候補を見つけやすい．

5.3 Krinke の手法

Krinke は，スライスベースでステートメント単位の凝集度メトリクスを提案している [14]. この凝集度メトリクスは式 2 で定義される．なお，SL_x は出力変数 x に対するスライスである．

$$Cohesiveness_{SL}(s) = \frac{\sum_{x|s \in SL_x} |SL_x|}{\sum_{x \in V_O} |SL_x|} \tag{2}$$

このメトリクスは，スライスを計算するためにプログラム依存グラフを作成する必要があり処理に少し時間がかかるが，ステートメント単位で凝集度を求めることができるので，本手法のブロック単位の凝集度に比べて精度が高い．

また，ステートメント単位の凝集度の可視化を行うことによって，メソッドの再構成をすべき箇所を示している．

6　まとめと今後の課題

本研究では，ソースコードブロック間の凝集度を用いることによって，政井らの手法の改善を行った．具体的には，政井らの手法によって提示される候補の抽出コード片を凝集度の大きさによって順位付けを行うことで，利用者にとって有用な候補を見つけやすくした．また，非常に多くの候補が現れる場合も，上位の結果だけを表示することで Eclipse 上のウィザード表示時にかかる時間の削減，リソースの節約が可能である．

Java で記述されたオープンソースソフトウェアのソースコードを対象に，被験者を用いて適用実験を行った結果，提案手法の有効性を確認できた．

今後の課題は，COB 以外のメトリクスを使用して候補の順位付けを行うことや，FTMPATool を 3 つ以上の類似メソッドの入力に対応することである．

提案手法では，計算量が少ない COB を用いた．しかし，COB はソースコードブロック間での協調関係のみに関与しているので，ステートメント単位の協調関係は分からない．そこで，プログラムスライスベースの凝集度や，プログラム依存グラフのデータ依存辺，制御依存辺を用いることによって，ステートメント単位での協調関係を利用することが考えられる．ステートメント単位での協調関係を利用すれば，ブロック間での協調関係を見ただけでは検出できなかった優れた候補を検出可能である．

FTMPATool では，3 つ以上の類似メソッドを入力として与えることができない．しかし，共通の親クラスを持つ 3 つ以上の子クラスが類似メソッドを持つ場合もあり，これらすべてに Template Method パターンを適用することができれば，さらなるプロジェクトの保守性の向上につながると考えられる．3 つ以上に対応する場合，候補数がさらに膨れ上がることが予想できるので，本手法を用いた順位付けを行い，上位のほうのみを提示することが必要である．このことを実現するために，FTMPATool に本手法を実装することも課題である．

謝辞

本研究において，常に適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より深く感謝いたします．

本研究において，随時適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に深く感謝いたします．

本研究において，逐次適切な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻石尾隆助教に深く感謝いたします．

本研究において，適時適切な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻肥後芳樹助教に深く感謝いたします．

本研究において，終始適切な御指導及び御助言を頂きました奈良先端科学技術大学院大学情報科学研究科ソフトウェア設計学講座吉田則裕助教に深く感謝いたします．

本研究において，様々な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻齋藤晃氏，山田吾郎氏，政井智雄氏に深く感謝いたします．

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします．

参考文献

[1] Apache Ant. `http://ant.apache.org/`.

[2] ANTLR parser generator. `http://www.antlr.org/`.

[3] Azureus, now called vuze : Bittorrent client. `http://azureus.sourceforge.net/`.

[4] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of ICSE '98*, pp. 368–377, Kyoto, Japan, 1998.

[5] Eclipse. `http://eclipse.org/`.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[8] 肥後芳樹. Scorpio. `http://sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio/`.

[9] 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.

[10] N. Juillerat and B. Hirsbrunner. Toward an Implementation of the "Form Template Method "Refactoring. In *Proc. of SCAM 2007*, pp. 81–90, Paris, France, 2007.

[11] 兼光智子, 肥後芳樹, 楠本真二. プログラム依存グラフを用いたリファクタリング候補の特定と可視化. 電子情報通信学会技術研究報告, Vol. 110, No. 336, pp. 61–66, 2010.

[12] R. K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-Based Reverse-Engineering of Design Components. In *Proc. of ICSE '99*, pp. 226–235, Los Angeles, CA, USA, 1999.

[13] J. Kerievsky. *Refactoring to Patterns.* Addison Wesley, 2004.

[14] J. Krinke. Statement-Level Cohesion Metrics and their Visualization. In *Proc. of SCAM '07*, pp. 37–48, Paris, France, 2007.

[15] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proc. of ICSM' 97*, pp. 314–321, Bari, Italy, 1997.

[16] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.*, Vol. 32, No. 3, pp. 176–192, 2006.

[17] 政井智雄, 吉田則裕, 松下誠, 井上克郎. テンプレートメソッドの形成に基づく類似メソッド集約支援. 日本ソフトウェア科学会 FOSE2010 ソフトウェア工学の基礎 XVII, pp. 125–130, 2010.

[18] 三宅達也, 肥後芳樹, 井上克郎. メソッド抽出の必要性を評価するソフトウェアメトリックスの提案. 電子情報通信学会論文誌 D, Vol. J92-D, No. 7, pp. 1071–1073, 2009.

[19] R. B. Yates and B. R. Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

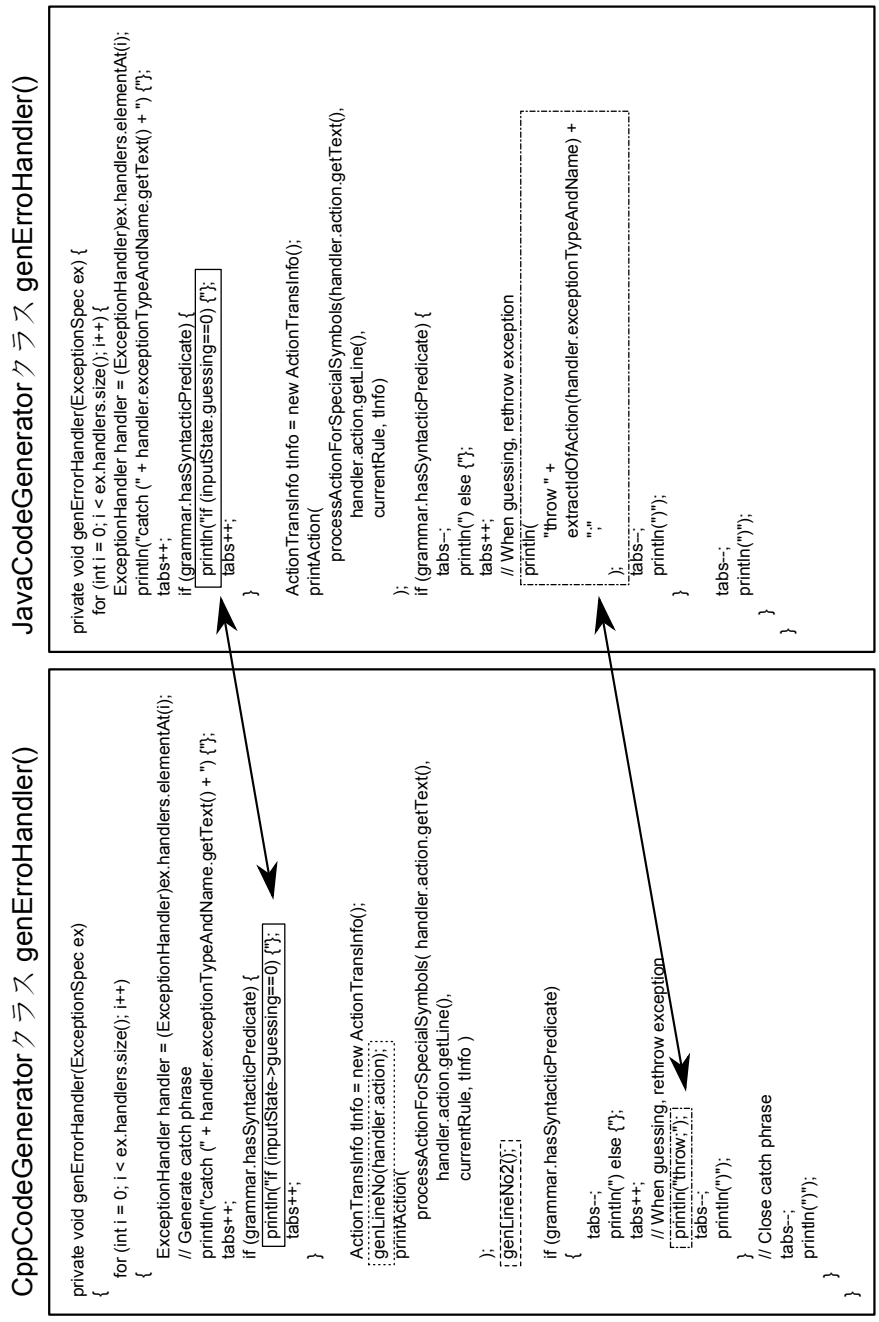
[20] A. Zeller. *Why Programs Fail*. Morgan Kaufmann Pub., 2005.

付録

A　実験対象メソッド

実験で使用したメソッドを掲載する．なお，四角で括られている箇所は差分である．

Antlr プロジェクト



Ant プロジェクト

Arc クラス executeDrawOperation()
public PlanarImage executeDrawOperation() {
BufferedImage bi = new BufferedImage(width + (stroke_width * 2),
height + (stroke_width * 2), BufferedImage.TYPE_4BYTE_ABGR_PRE);
Graphics2D graphics = (Graphics2D) bi.getGraphics();
if (!stroke.equals("transparent")) {
BasicStroke bStroke = new BasicStroke(stroke_width);
graphics.setColor(ColorMapper.getColorByName(stroke));
graphics.setStroke(bStroke);
graphics.draw(new Arc2D.Double(stroke_width, stroke_width, width,
height, start, stop, type));
}
if (!fill.equals("transparent")) {
graphics.setColor(ColorMapper.getColorByName(fill));
graphics.fill(new Arc2D.Double(stroke_width, stroke_width,
width, height, start, stop, type));
}
for (int i = 0; i < instructions.size(); i++) {
ImageOperation instr = ((ImageOperation) instructions.elementAt(i));
if (instr instanceof DrawOperation) {
PlanarImage img = ((DrawOperation) instr).executeDrawOperation();
graphics.drawImage(img.getAsBufferedImage(), null, 0, 0);
} else if (instr instanceof TransformOperation) {
graphics = (Graphics2D) bi.getGraphics();
PlanarImage image = ((TransformOperation) instr)
.executeTransformOperation(PlanarImage.wrapRenderedImage(bi));
bi = image.getAsBufferedImage();
}
}
return PlanarImage.wrapRenderedImage(bi);
}
Ellipse クラス executeDrawOperation()
public PlanarImage executeDrawOperation() {
BufferedImage bi = new BufferedImage(width,
height, BufferedImage.TYPE_4BYTE_ABGR_PRE);
Graphics2D graphics = (Graphics2D) bi.getGraphics();
if (!stroke.equals("transparent")) {
BasicStroke bStroke = new BasicStroke(stroke_width);
graphics.setColor(ColorMapper.getColorByName(stroke));
graphics.setStroke(bStroke);
graphics.draw(new Ellipse2D.Double(0, 0, width, height));
}
if (!fill.equals("transparent")) {
graphics.setColor(ColorMapper.getColorByName(fill));
graphics.fill(new Ellipse2D.Double(0, 0, width, height));
}
for (int i = 0; i < instructions.size(); i++) {
ImageOperation instr = ((ImageOperation) instructions.elementAt(i));
if (instr instanceof DrawOperation) {
PlanarImage img = ((DrawOperation) instr).executeDrawOperation();
graphics.drawImage(img.getAsBufferedImage(), null, 0, 0);
} else if (instr instanceof TransformOperation) {
graphics = (Graphics2D) bi.getGraphics();
PlanarImage image = ((TransformOperation) instr)
.executeTransformOperation(PlanarImage.wrapRenderedImage(bi));
bi = image.getAsBufferedImage();
}
}
return PlanarImage.wrapRenderedImage(bi);
}

Azureus プロジェクト