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

信学技報
IEICE Technical Report
SS2008-25 (2008-07)

ソフトウェアメトリクスとメソッド内の構造を用いたリファクタリング支援手法の提案

三宅　達也† 　肥後　芳樹† 　井上　克郎†

† 大阪大学大学院情報科学研究科
〒 560–8531 大阪府豊中市待兼山町 1–3
E-mail: †{t-miyake,higo,inoue}@ist.osaka-u.ac.jp

あらまし　リファクタリングはソフトウェアの外部的振る舞いを保ったまま，内部構造を変更することにより，ソフトウェアの設計品質を改善する．しかしながら，リファクタリングは複数の手順で行われる複雑な作業である．このため手動によるリファクタリングは誤りが発生しやすく，適切に行うには高い技術と多大な時間を必要とする．そこで本研究ではリファクタリング手順の一部を自動化する手法を提案する．具体的には，リファクタリング手順の重要な要素である「リファクタリングすべき箇所の識別」,「適用すべきリファクタリングパターンの決定」,「リファクタリングの効果予測」を自動化する．

キーワード　リファクタリング, 自動化, ソフトウェアメトリクス

A Refactoring Approach Based on Software Metrics and Internal Structure of Method

Tatsuya MIYAKE†, Yoshiki HIGO†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1–3 Machikaneyama-cho, Toyonaka, Osaka, 560–8531 Japan
E-mail: †{t-miyake,higo,inoue}@ist.osaka-u.ac.jp

Abstract　Refactoring is a set of operations to improve design quality of software without changing the external behavior of it. However, refactoring is a complicated task. Manual refactoring is error-prone, and it requires certain costs to be performed appropriately. This paper proposes an approach that automatizes a part of refactoring operations. The approach identifies where should be refactored, and it suggests how they should be improved. Moreover, the approach estimates the effect of the refactorings.

Key words　Refactoring, Automation, Software Metrics

1. はじめに

ソフトウェアの設計品質はソフトウェアの開発や保守を効率的に行うための重要な因子である．近年，ソフトウェアは社会のさまざまな分野で重要な役割を果たしており，より利便性や信頼性の高いソフトウェアが要求されている．しかし，ソフトウェアの利便性を高めるため大規模化，複雑化したソフトウェアの信頼性を高く保つには，開発や保守に大きなコストを必要とする．このため，ソフトウェアの設計品質の重要度はますます高まっている．

しかし，大規模ソフトウェアの開発プロジェクトでは複数のプログラマが，さまざまな要求を満たすようにソフトウェアを開発，修正していくため，ソフトウェアの設計品質を高く保つことは難しい．このような問題に対処するために，ソフトウェアの設計品質を向上させる技術の 1 つであるリファクタリングが注目されている．

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 [2]．しかしながら，手動によるリファクタリングは次の問題を抱えているため，実際に開発現場で用いられることは稀である．

- 大規模ソフトウェアから手動でリファクタリングすべき箇所を特定するのは非常に手間がかかる．
- 特定した箇所をどのようにリファクタリングすべきか決定するには高い技術力が必須である．
- リファクタリングの効果を事前に予測することは難しい

ため，必要としたコストに効果が見合わない可能性がある．

- 手動による変更は誤りが混入しやすく，開発者に多くのテストを強要する．

これらの問題から開発者を解放するためには，リファクタリング作業を自動化する手法が必要である．

本研究では，リファクタリング作業の一部を自動化する手法を提案する．提案手法はメソッド抽出リファクタリングとそこから派生するリファクタリングを自動化する．具体的には，ソフトウェアメトリクスを用いてメソッドとして抽出すべきコード片の基点となる位置を特定した後，その周囲のブロック文構造や変数の使用状況に基づいて適切な抽出範囲と適用すべきリファクタリングパターンを決定する．さらに，リファクタリング前後におけるソフトウェアメトリクスの変化を予測することにより，リファクタリング適用前にその効果を予測する．

以下，2章ではリファクタリングの手順と本手法で自動化されるリファクタリングパターンについて述べる．3章では提案手法に関して説明し，4章でまとめと今後の課題について述べる．

2. リファクタリング

リファクタリングとはソフトウェアの保守性や可読性，再利用性などを向上させるために，ソフトウェアの外部的振る舞いを保ちつつ，内部的構造を変化させる一連の作業を意味する [2]．

2.1 リファクタリングプロセス

リファクタリングは次の5つのプロセスに分けて行われる．

プロセス1 ソフトウェアの品質が低くリファクタリングすべきである箇所 (以後，不吉な匂いと呼ぶ) を識別する．不吉な匂いの代表的な例として，長すぎるメソッドや複雑な制御構造，重複コードなどが挙げられる [2]．

プロセス2 識別した箇所をどのように修正すべきか検討する．既存の研究において，不吉な匂いの種類に対応したさまざまなリファクタリングパターンが提案されている [2]．

プロセス3 リファクタリングの効果とコストを予測する．これにより，コストに見合わないリファクタリングを回避することができる．

プロセス4 プロセス2で決定したリファクタリングパターンに基づいて内部構造を再構築する．

プロセス5 再構築されたソフトウェアが外部的振る舞いを保っているか検証する．

2.2 リファクタリングパターン

本研究で半自動化されるリファクタリングパターンの概要を説明する．

2.2.1 Extract Method

提案手法による半自動化の核とな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は長すぎるメソッドからその一部を新しいメソッドとして抽出することにより，短いメソッドに分割する．短いメソッドは次の利点をもつ [2]．

- 他のメソッドから利用できる可能性が高い (再利用性が高い) ため，重複コードの生成防止につながる．
- 適切な命名を行うことで内部の処理の理解が容易になる．また，呼び出しもとのメソッドをコメント列のように読むことができ，可読性が高まる．
- オーバーライドが行いやすく，拡張性が高い．

また，Extract Method は他のさまざまなリファクタリングパターンの前準備として頻繁に利用される普遍的なリファクタリングパターンの1つである．Extract Method は効果的で頻繁に適用されるリファクタリングであるため，自動化の効果は大きく，既存の研究においてもさまざまな自動化手法やツールが提案されている [1], [5], [6]．

2.2.2 Move Method

あるメソッドが不適切なクラスで宣言されている場合，Move Method リファクタリングを用いて適切なクラスに移動することにより，クラスが持つ責任を明確化する．このリファクタリングは単独でも行われるが，Extract Method から派生して行われることもある [2]．例えば，メソッドの一部分のみが不適切なクラスに記述されている場合，該当部分を Extract Method により新規メソッドとして抽出し，その後，Move Method を用いて適切なクラスに移動する．

本手法でも，Extract Method により抽出されたメソッドを適切なクラスに移動するためにこのリファクタリングパターンを利用する．

2.2.3 Pull Up Method

同一の処理を行うメソッドが複数のサブクラスに存在する場合，Pull Up Method を用いて重複したメソッドをスーパークラスに引き上げ1つのメソッドにまとめることにより，保守性を向上させる．また，処理が重複していない場合も，対象となるメソッドがサブクラスの属性やメソッドを使用していなければ，事前にスーパークラスに引き上げておくことにより，そのメソッドが再利用される可能性は向上する．

本手法では，Extract Method の対象となる範囲内でサブクラスの属性やメソッドが利用されていない場合，このリファクタリングパターンを利用する．

3. 提案手法

本手法はメソッド抽出リファクタリングとそこから派生するリファクタリングを半自動化する．本手法は次のステップでリファクタリングを半自動化する．

（1） メトリクスを用いて不吉な匂いを特定
（2） 新規メソッドとして抽出すべき範囲を識別
（3） 新規メソッドの抽出先を決定
（4） メトリクスの変化を計測し，リファクタリングの効果を予測

これにより，リファクタリングプロセス1，2，3が半自動化される．また，本手法が提供するリファクタリングは単純なメソッド抽出であるが，それゆえに変更後のコードは制御フローやデータフローを変更前のまま維持しており，ホワイトボックステストの新しいテストケースを必要としない (プロセス5のコストが軽減される)．さらに，本手法を Eclipse [1] などの既存のツールと併用することにより，プロセス4も自動化することができ，すべてのプロセスを半自動化することが可能である．

本章では，まず本研究で提案する概念である「ローカル変数

のカプセル化」について述べ，次に本手法の詳細を各ステップごとに述べる．

3.1 ローカル変数のカプセル化

オブジェクト指向プログラミングにおいて，クラスの属性は外部からの干渉を避けるために隠蔽 (属性のカプセル化) される．カプセル化を進めることによりクラス内部の仕様変更による外部への影響を抑えることが可能となり，ソフトウェアの保守性や拡張性，再利用性などが向上する．このため，カプセル化はオブジェクト指向プログラムの品質を向上させるための重要な概念として広く知られている．

ここで，我々は属性のカプセル化と同じく，情報隠蔽の概念にもとづき,「ローカル変数のカプセル化」という概念を提案する．ローカル変数のカプセル化とはあるコード片において利用可能であるが利用する必要のないローカル変数をそのコード片から不可視にすることである．クラスの属性を隠蔽する場合，その属性の宣言にアクセス制約を記述することによりアクセスを制限し，必要に応じてアクセサメソッドを用意する．一方，ローカル変数を隠蔽する場合は，そのローカル変数を不可視にしたいコード片を新規メソッドとして抽出する．そして，抽出されたコード片の外側で宣言されているが可視のままにしておきたい変数のみ引数や返り値を用いてアクセスする．これにより抽出されたコード片内の変更による影響を，抽出もとのメソッドで考慮する必要がなくなる．また，抽出もとのメソッド内の変更を抽出されたコード片側で考慮する必要がなくなる．これは属性のカプセル化と同様の効果であり，ソフトウェアの保守性や再利用性を向上させると考えられる．

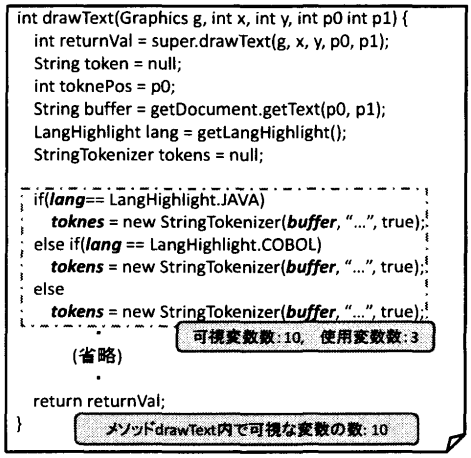
ローカル変数のカプセル化の例を図 1 に示す．図 1(a) のソースコードにおいて，枠線で囲まれた if-else 文内で可視変数の数は，パラメータ変数とローカル変数を合わせると 10 である．しかし，実際に使用されている変数は変数 $buffer$, $lang$, $tokens$ の 3 つのみであり，残りの 7 つの変数は使用されていない．このような場合，枠線で囲まれた if-else 文を含むコード片を新規メソッドとして抽出し，図 1(b) のソースコードのように変更することにより，枠線で囲まれた if-else 文内で可視である変数は使用されている変数のみとなり，使用する必要がない 7 つの変数に対する処理を考慮する必要がなくなる．このとき，枠線で囲まれた if-else 文内のみで使用されている変数 $lang$ は，変数宣言自体を新規メソッドとして抽出する範囲に含むことにより，抽出もとのメソッド $drawText$ から不可視になる．これによりメソッド $drawText$ 側で変数 $lang$ に対する処理を考慮する必要がなくなる．抽出もとのメソッドで宣言された変数を参照したい場合は，変数 $buffer$ のようにパラメータ変数を用いて参照することができる．抽出もとのメソッドで宣言された変数に代入したい場合は，変数 $tokens$ のように新規メソッドの返り値を用いて代入することができる．

3.2 (1) 不吉な匂いの特定

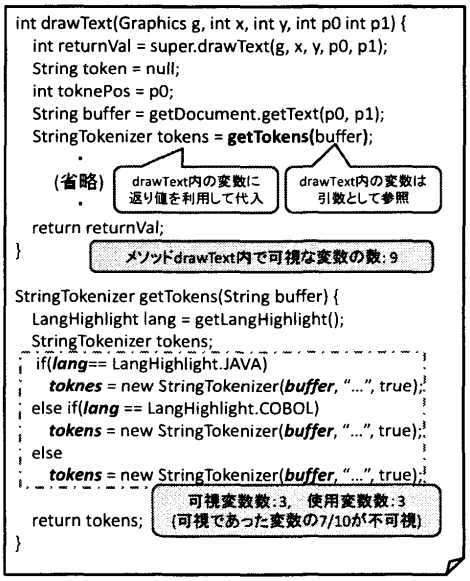
ソフトウェアメトリクスを計測して，その値をもとに抽出すべきブロック文を特定する．

計測されるメトリクスは次の 3 つである．

サイクロマチック数 サイクロマチック数とは，McCabe によって提案された複雑度メトリクスである [4]．プログラムの制御フローを有向グラフで表現したとき，開始ノードから終了ノードまでの経路の数で表される．この値は直観的にはソースコード上の分岐の数に 1 を加えた数を表す．サイクロマチック数が大きいほど制御フローは複雑になり，保守性や可読性が低いことを意味する．サイクロマチック数の高いコード片は，メソッドとして抽出することにより条件記述を分解できるため，Extract Method の対象に含まれる [2].



(a) メソッド抽出前



(b) 枠線部のメソッド抽出後

図 1 ローカル変数のカプセル化

一般にサイクロマチック数はメソッドの複雑度を表す尺度として用いられるが，本手法ではブロック文単位で計測する．ブロック文 B に含まれる条件文の数を n とすると， サイクロマチック数 $CC(B)$ は次の式で表される．

$$CC(B)(CyclomaticComplexity) = n + 1$$

抽出対象のコード片に含まれるブロック文のサイクロマチック数が高いほど，抽出もとのメソッドのサイクロマチック数の減少値が大きい (メソッド抽出の効果が大きい)．

メソッドのサイクロマチック数 $CC(M)$ と $CC(B)$ の比率

$RCC(B)$ を計算する．

$$RCC(B)(RatioofCyclomaticComplexity) = \frac{CC(B)}{CC(M)}$$

$RCC(B)$ が高すぎるとき，ブロック文 B を抽出しても条件記述は新しいメソッドに移動するだけで，分解されることはない．

以上のことをふまえ，次の条件を満たすブロック文 B を含むコード片をメソッド抽出の対象とする．

$$(CC(B) >= T_{cc}) \ \wedge \ (RCC(B) <= T_{rcc})$$
$$(T_m : \text{メトリクス } m \text{ の閾値})$$

LOC(Lines Of Code) LOC はソースコードの行数を表す．一般に LOC が高いメソッドは可読性が低く，保守が行いにくい．このため，LOC が高いメソッドは Extract Method の対象となる不吉な匂いの代表として知られている [2]．

本手法ではブロック文単位の LOC と，そのブロック文のメソッド内における占有率を計測する．メソッド M の LOC を $LOC(M)$，メソッド M に含まれるブロック文 B の LOC を $LOC(B)$，とするとブロック文 B の占有率 $RLOC(B)$ は次の式で表される．

$$RLOC(B)(RatioofLinesOfCode) = \frac{LOC(B)}{LOC(M)}$$

$RLOC(B)$ が高すぎるとき，新しいメソッドが抽出もとのメソッドの問題 (LOC が高い) を継承してしまうため，リファクタリングの効果は低い．

これらのことをふまえ，次の条件を満たすブロック文 B を含むコード片をメソッド抽出の対象とする．

$$LOC(B) >= T_{loc} \ \wedge \ RLOC(B) <= T_{rloc}$$
$$(T_m : \text{メトリクス } m \text{ の閾値})$$

LVA(Local Variables Availability) このメトリクスは本論文で提案する新しいメトリクスである．LVA はあるブロック文内で利用可能なローカル変数の数と実際に利用されている変数の数の比率を表す．ブロック文 B 内で使用可能な変数の集合を $AV(B)$，実際にブロック文 B 内で使用されている変数の集合を $UV(B)$ とすると，$LVA(B)$ は次の式で表される．

$$LVA(B) = \frac{|UV(B)|}{|AV(B)|}$$

ローカル変数のカプセル化が行われていないメソッドは，その内部のあるブロック文において利用可能であるが利用されていない変数が多くなる．すなわち LVA の値が小さくなる．このため，LVA の値が閾値よりも小さいブロック文を含むコード片を，メソッド抽出の対象とする．

3.3 (2) 抽出べき範囲を識別

抽出箇所の特定により，メソッドとして抽出することでメトリクスの値が改善するブロック文が特定される．しかし，本来，メソッドの抽出はメトリクスの値を改善させるためではなく，機能ごとに分割することを目的に行われる．そこで本手法では特定されたブロック文を含む機能的範囲を識別し，それをメソッドとして抽出すべき範囲とする．これにより，効果的 (メトリクスが改善) かつ適切 (機能に基づいた分割) なメソッド抽出リファクタリングを行える．

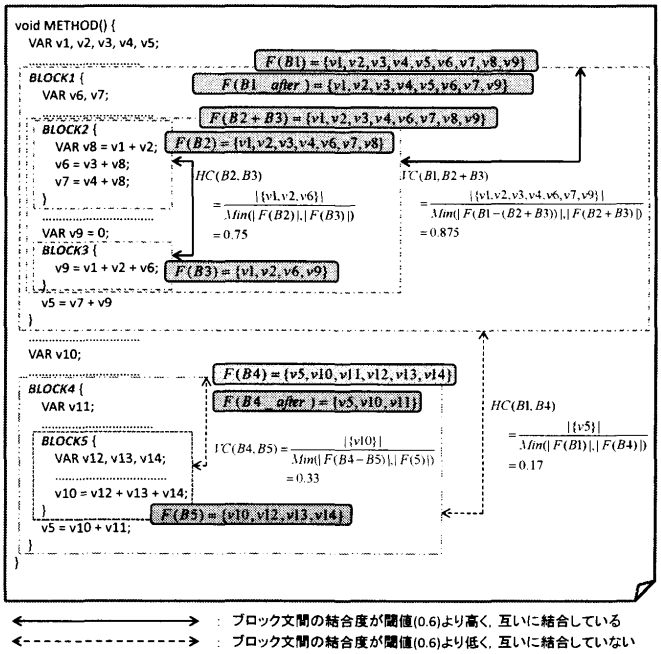


図 2 ブロック文間の結合度を用いた機能分割

ここで，メソッドの機能はそのコード内に現れる変数の値を決定する計算であると捉えることができる．つまり，コード内の特定の機能を指定するには，その機能に対応した変数を指定すればよい [6]．ただし，メソッド内の処理は複数の変数を用いた計算により成立する．これらのことを考慮すると n 個のローカル変数が内部で使用されているメソッド M の機能 $F(M)$ は次のように変数の集合で表すことができる．

$$F(M) = \{v_1, v_2, \cdots v_n\}(v_i : M \text{ 内で使用されている変数})$$

同様に考えると，メソッド M 内のブロック文 B の機能 $F(B)$ はブロック文 B 内で使用されている変数の集合で表すことができる．このとき，2 つの集合 $F(M)$，$F(B)$ 間には $F(B) \subset F(M)$ が成り立ち，機能 $F(B)$ は機能 $F(M)$ の一部であると判断できる．ブロック文 B はメソッド M の内部のコードであるため，この判断の妥当性は高い．

このことを踏まえ，メトリクスに基づいて特定されたブロック文と同一の機能に属するブロック文および単文 (宣言文や代入文など) を識別する．

3.3.1 ブロック文間の結合

次の 2 つの結合度を用いて 2 つのブロック文が同一の機能に属するか否か識別する．結合度が大きいほど 2 つのブロック文が同一の機能に属する可能性が高いと判断する．

水平結合度 水平の関係を持つブロック文間の結合度を表す．ここで水平の関係とは，同一のブロック文の直下にある 2 つブロック文間の関係を意味する．例えば，図 2 の BLOCK1 と BLOCK4，BLOCK2 と BLOCK3 は互いに水平の関係を持つ．あるブロック文 B_1 の機能を $F(B_1)$，もう一方のブロック文 B_2 の機能を $F(B_2)$ とすると，2 つのブロック文の水平結合度

$HC(B_1, B_2)$ は次の式で表される．$|F(B)|$ は機能 $F(B)$ を構成する変数の数を表す．

$$HC(B_1, B_2) = \frac{|F(B_1) \cap F(B_2)|}{Min(|F(B_1)|, |F(B_2)|)}$$

$(Min(A, B)$： A と B のうち小さい方の値を示す$)$

$HC(B_1, B_2)$ が大きいとき，$F(B_1)$ と $F(B_2)$ のうち一方もしくは両方の要素の大部分がもう一方の集合に含まれる．特に $HC(B_1, B_2)$ の値が 1(最大値) のとき，$F(B_1) \subset$ (もしくは $\supset) F(B_2)$ が成り立つ．つまり，$HC(B_1, B_2)$ の値が閾値より大きければ，一方のブロック文の機能はもう一方のブロック文の機能の一部，もしくは，類似した機能であると判断できる．このため，3.2 節で特定されたブロック文との水平結合度が大きいブロック文が存在する場合，そのブロック文もメソッド抽出の範囲に含まれる．

水平結合度の閾値を 0.6 としたときの例を図 2 に示す．ブロック文 BLOCK2 内では 7 つの変数が使用されており，機能 $F(B_2)$ は $v1, v2, v3, v4, v6, v7, v8$ で表される．同様にブロック文 BLOCK3 の機能 $F(B_3)$ は $v1, v2, v6, v9$ で表される．このとき，$F(B_2) \cap F(B_3)$ は $v1, v2, v6$(要素数 3)，$Min(|F(B_2)|, |F(B_3)|)$ は 4 となる．ゆえに，水平結合度 $HC(B_2, B3)$ は 0.75 となり閾値より高いため，BLOCK2 と BLOCK3 は同一の機能に属すると判断する．また，BLOCK1 と BLOCK4 の水平結合度は 0.17 であり，閾値より低いため 2 つのブロック文は異なる機能に属すると判断する．

垂直結合度　垂直の関係を持つブロック文間の結合度を示す．ここで垂直の関係とは，あるブロック文 B_{in} とその外側のブロック文 B_{out} 間の関係を意味する．例えば，図 2 の BLOCK1 と BLOCK2，BLOCK1 と BLOCK3，BLOCK4 と BLOCK5 は互いに垂直の関係にある．ここで，B_{in} の機能を $F(B_{in})$，B_{out} の機能を $F(B_{out})$ とする．また，ブロック文 B_{in} 内で宣言している変数群を $DV(B_{in})$ とする．このとき，ブロック文 B_{in} がメソッドとして抽出されると仮定すると，抽出後の外側のブロック文 B_{out_after} にとって，内側のブロック文 B_{in} 内で宣言していた変数群 $DV(B_{in})$ は不可視になる． このため，機能 $F(B_{out\_after})$ は次の式で表される．

$$F(B_{out\_after}) = F(B_{out}) - DV(B_{in})$$

抽出もとのメソッドに残るブロック文の機能 $F(B_{out\_after})$ と新規メソッドとして抽出されたブロック文の機能 $F(B_{in})$ が類似している場合，同一の機能が複数のメソッドにまたがって実装されることになる．このため，機能に基づいた分割を実現するには，内側のブロック文 B_{in} のみでなく，外側のブロック文 B_{out} ごと抽出する必要がある．以上のことを考慮し，2 つのブロック文 B_{in}，B_{out} の垂直結合度 $VC(B_{in}, B_{out})$ は抽出もとのメソッドに残るブロック文の機能 $F(B_{out\_after})$ と新規メソッドとして抽出されたブロック文の機能 $F(B_{in})$ の類似度で表す．具体的には次の式で表される．

$$VC(B_{in}, B_{out}) = \frac{|F(B_{out\_after}) \cap F(B_{in})|}{Min(|F(B_{out\_after})|, |F(B_{in})|)}$$

$VC(B_{in}, B_{out})$ が大きいとき，水平結合が大きいときと同様の理由で，内側のブロック文の機能は外側のブロック文の機能の一部，もしくは，類似した機能であると判断できる．このため，3.2 節で特定されたブロック文と外側のブロック文の垂直結合度が閾値より大きい場合，外側のブロック文もメソッド抽出の範囲に含まれる．

垂直結合度は水平の関係を持つ複数のブロック文群とそれらの外側のブロック文に対しても計測することができる．この場合，内側のブロック文群の機能は各ブロック文の機能を表す集合の和集合で表される．

垂直結合度の閾値を 0.6 としたときの例を図 2 に示す．ブロック文 BLOCK5 内では 4 つの変数が使用されており，機能 $F(B_5)$ は $v10, v12, v13, v14$ で表される．BLOCK5 を新規メソッドとして抽出する場合，BLOCK5 で使用されている変数のうち，BLOCK5 内で宣言されている変数 $v12$，$v13$，14 は BLOCK4 から不可視になる．変数 $v10$ のみが，新規メソッドの引数として BLOCK4 内で使用され続ける．このため，BLOCK5 をメソッドとして抽出した後の BLOCK4 の機能 $F(B_{4\_after})$ は $v5, v10, v11$ で表される．このとき，$F(B_{4\_after}) \cap F(B_5)$ は $v10$(要素数 1)，$Min(|F(B_{4\_after})|, |F(B_5)|)$ は 3 となる．ゆえに，垂直結合度 $HC(B_4, B5)$ は 0.33 となり閾値より低いため，BLOCK4 と BLOCK5 は異なる機能に属すると判断する．また，BLOCK2 と BLOCK3 を含むコード片の機能 $F(B_2 + B_3)$ は $F(B_2) \cup F(B_3)$ で表される．このため，BLOCK2 と BLOCK3 を含むコード片と BLOCK1 の垂直結合度は 0.875 となり閾値より高いため，BLOCK2 と BLOCK3 を含むコード片と BLOCK1 は同一の機能に属すると判断する．

3.3.2　同一機能に属するブロック文群の識別

3.2 節で特定されたブロック文と同一の機能に属すると考えられるブロック文群をブロック文間の結合を用いて，次のアルゴリズムで識別する．

アルゴリズム

```
1. RESULT = NULL
2. B = 3.2 節で特定されたブロック文
3. HCB = B と水平結合しているブロック文群 (B を含む)
4. NHCB = B と水平結合していないブロック文群
5. OB = B の外側のブロック
6. if (NHCB が空でない)
7.   if (VC(OB,HCB) > VC(OB,NHCB))
8.     NHCB を新規メソッドとして抽出
9.   else
10.    HCB を新規メソッドとして抽出
11.    EXIT
12.  end if
13. end if
14. if(OB と HCB が垂直結合している)
15.   if(OB がメソッド全体でない)
16.     B = OB
17.     GOTO 行 3
18.   end if
19. else
20.   HCB を新規メソッドとして抽出
21. end if
```

このアルゴリズムを図 2 のコードに適用した場合の例を示す．水平結合と垂直結合の閾値はどちらも 0.6 とする．3.2

節でメトリクスを用いてメソッドとして抽出すべき箇所として BLOCK2 が特定される (行 2) と仮定する．このとき BLOCK2 は BLOCK3 と水平結合しており (行 3)，BLOCK1 内に BLOCK2 と水平結合していないブロック文は存在しない (行 4,5,6)．BLOCK2 と BLOCK3 を含むコード片は BLOCK1 と垂直結合している (行 14)．BLOCK1 はメソッド全体ではなくメソッド内のブロックの 1 つであるため，他のブロックとの結合を調べる (行 16,17)．BLOCK1 は BLOCK4 と水平結合しておらず (行 4,6)，外側のブロックであるメソッド本体とは BLOCK1 の方が強く垂直結合している (行 7) ため，BLOCK4 を新規メソッドとして抽出する (行 8)．残った BLOCK1 は外側のブロックであるメソッド本体と垂直結合している (行 14,15) ため，もとのメソッド METHOD 内に残る (行 18,21)．以上の処理によりメソッド METHOD は BLOCK1 を含むコード片と BLOCK4 を含むコード片に分割される．

3.3.3 同一の機能に属する周囲のコードの識別

3.3.2 節のアルゴリズムでメソッドとして抽出すべきブロック文群が識別される．しかし，より精度の高い機能分割を実現するためには，それぞれのブロック文の前後の文も，ブロック文と同一の機能に属するようであれば，メソッド抽出範囲として識別する必要がある．本手法では，簡単なプログラムスライシングを用いて識別する．スライシングの基点にはブロック文の外側で宣言さている変数 (以降，外部変数と呼ぶ) のうち，ブロック文と関連が強い変数を指定する．ブロック文と外部変数の関連の強さは，その外部変数がブロック文の内外のどちらで頻繁に使用されているかで決定される．ブロック文の内部で頻繁に使用されている外部変数ほど，そのブロック文と関連が強いと判断する．ただし，メソッドの機能はそのコード内に現れる変数の値を決定する計算であると捉えることができるため，変数がブロック文の内部のみで代入されている場合は，外部で頻繁に参照されていても内部で代入を行っているブロック文と関連が強いと判断する．この際，変数の初期化に関しては，代入であるとは見なさない．これらのことを考慮すると，図 2 の変数 $v9$ は BLOCK3 の外側で宣言，初期化されているが，ブロック文の内部で値が決定されているため，BLOCK3 と関連の強い外部変数であると判断され，スライスの基点とする．

3.4 (3) 新規メソッドの抽出先を決定

抽出対象のコード片内で使用されている属性や呼び出されているメソッドをもとに，抽出対象のコード片をどのクラスのメソッドとして抽出すべきか決定する．

- 抽出対象のコード片内で自クラスの属性やメソッドが主に使用されている場合，自クラスのメソッドとして抽出する．
- 自クラスと継承関係がないクラスの属性やメソッドを主に使用している場合，Move Method を使用し，属性やメソッドが最も多く使用されているクラスのメソッドとして抽出する．
- 抽出対象のコード片内でスーパークラスの属性やメソッドが主に使用されており，かつ，自クラスの属性やメソッドがまったく使用されていない場合，Pull Up Method を使用しスーパークラスのメソッドとして抽出する．

3.5 (4) ファクタリングの効果を予測

リファクタリングの効果を示す尺度としてメトリクスの変化を利用することができる [3]．本手法では 3.2 節で記述したメトリクスの変化量をリファクタリングの効果とする．3.2 節で記述したメトリクスの変化は，基本的には抽出対象となる部分のメトリクスに等しい．

また，抽出されたメソッドを他のクラスに移動した場合，クラス間の結合度が変化する可能性がある．このため，Move Method を利用する場合は，リファクタリング前後のクラス間の結合度の変化も測定する．

これらのメトリクスの変化をもとにリファクタリングの効果と副作用を予測し，リファクタリングを行うか否か決定する．

4. まとめと今後の課題

本論文ではリファクタリング作業の一部を自動化する手法を提案した．本手法は既存のオブジェクト指向プログラムから新規メソッドとして抽出すべきコード片を特定し，さらに，そのコード片の適切な抽出先を決定する．さらに，リファクタリングの効果をコード変更前に予測するため，ユーザは効果的なリファクタリングのみを選択することができる．

今後の課題としては次の内容があげられる．

- 本研究ではメソッドの機能を変数の集合で表現している．しかし，メソッドの機能を実装するうえで，各変数の重要度は異なる．例えば，使用頻度の高い変数や返り値となる変数は，他の変数と比べて重要度が高いと考えられる．このことを考慮し，各変数に応じた重みを負荷したうえで，LVA やブロック文間の結合度を計測することにより，自動化されたリファクタリングがより妥当になると考えられる．
- 本研究では新しいメトリクスである LVA を提案している．本論文ではオブジェクト指向プログラミングのカプセル化の概念をもとに，定性的な観点から LVA の有用性を評価しているが，定性的な評価のみでは必ずしも有用であるとは言えない．そこで，LVA とバグの相関などを調査することにより，LVA がソフトウェアの品質に与える影響を定量的に調査する必要がある．

文　献

[1] Eclipse. `http://www.eclipse.org`.

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

[3] Y. Higo, Y. Matsumoto, S. Kusumoto, and K. Inoue. Refactoring Effect Estimation based on Complexity Metrics. In *Proc. of the 19th Australian Software Engineering Conference*, pp. 219–228, March 2008.

[4] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec 1976.

[5] E. Murphy-Hill and A. P. Black. Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In *Proc. of the 30th Intenational Conference on Software Engineering*, pp. 421–430, May 2008.

[6] 丸山. 基本ブロックスライシングを用いたメソッド抽出リファクタリング. 情報処理学会論文誌, 43(6):1625–1637, Jun 2002.