修士学位論文

題目

同一メソッド内に含まれるブロック間の結合度を用いたメソッド分割手法の提案

指導教員

井上 克郎 教授

報告者

三宅 達也

平成 21 年 2 月 9 日

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 ソフトウェア工学講座

平成 20 年度 修士学位論文

同一メソッド内に含まれるブロック間の結合度を用いた
メソッド分割手法の提案

三宅 達也

内容梗概

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．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．また，どのようなときリファクタリングが必要であるかを示す厳密な基準は存在しない．このため手動によるリファクタリングは多くの知識や経験を必要とする．また，熟練した開発者であっても大規模ソフトウェアに対してリファクタリングを行うには多大な時間を必要とする．そこで本研究では代表的なリファクタリングパターンの 1 つであるメソッド抽出リファクタリングを支援する手法を提案する．具体的には，メソッド内の構成要素間の協調度合を示すソフトウェアメトリクスを提案し，それを用いてメソッド抽出リファクタリングを必要とするメソッドの候補を検出する．さらに，メソッド内のコードブロック間の結合度を用いて，新規メソッドとして抽出すべき範囲を識別する．適用事例では提案したメトリクスを用いることにより，既存のメトリクスでは検出できないメソッド抽出リファクタリングの候補を検出できることを確認した．また，本手法に基づいたメソッド抽出リファクタリングの例を用いて，本手法が適切なメソッド抽出範囲の特定を支援することを示した．

主な用語

リファクタリング
メソッド抽出
ソフトウェアメトリクス
凝集度
プログラムスライシング

目次

1 まえがき **4**

2 背景 **6**

2.1 メソッド抽出リファクタリング 6

2.1.1 概要 6

2.1.2 メソッド抽出手順 6

2.1.3 メソッド抽出リファクタリングを必要とするメソッド 7

2.1.4 適切なメソッド抽出 7

2.1.5 メソッド抽出の利点 8

2.2 プログラムスライシング 8

2.3 凝集度 9

2.4 ソフトウェアメトリクス 10

3 ソフトウェアメトリクス: COB **13**

4 コードブロック間の結合度を用いたメソッド抽出範囲の識別 **15**

4.1 変数を用いた機能の抽象化 16

4.2 コードブロック間の関係 16

4.3 コードブロック間の結合 17

4.4 同一機能に属するコードブロック群の識別 21

4.5 同一の機能に属する周囲のコードの識別 22

5 実装 **24**

5.1 COB 計測プラグイン 24

5.2 メソッド抽出範囲識別ツール 25

6 適用実験 **27**

6.1 COB を用いたメソッド抽出候補の検出 27

6.2 SBC/PBC を用いたメソッド抽出範囲の識別事例 29

7 考察 **32**

8 関連研究 **33**

9 むすび **35**

謝辞 **36**

参考文献 **37**

1　まえがき

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．一方で，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 [3]．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．

- どのようなときリファクタリングが必要であるかを示す厳密な基準が存在しないため，この判断には多くの経験や知識を必要とする [3].
- 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多くの経験や知識を必要とする [3].

これらの問題を軽減するためには，リファクタリング作業を支援する手法が必要である．本研究では，代表的なリファクタリングパターンの 1 つであるメソッド抽出リファクタリングを支援する手法を提案する．メソッド抽出とは，既存のメソッドから適度な規模のコード片を新しいメソッドとして抽出する作業を意味する．メソッド抽出リファクタリングを必要とするメソッドの候補として，長すぎるメソッドや制御ロジックが複雑なメソッドなどがあげられるが [3, 8]，本来メソッドは行数や複雑さではなく，機能に基づいて分割することが望ましい [21]．ここで述べる機能とは 1 つのメソッドが実現する特定の処理のことを意味する (以降，機能という言葉を同様の意味で扱う)．メソッドの機能はメソッド内の構成要素が協調しあって実現される．そこで本研究では，メソッド内の構成要素間の協調度合を示すソフトウェアメトリクスを提案し，それを用いてメソッド抽出リファクタリングを必要とするメソッドの候補を検出する．また，メソッド内のコードブロック間の結合度を用いて，新しいメソッドとして抽出すべき範囲を特定する．

以下，2 章では本研究の背景としてメソッド抽出リファクタリング，プログラムスライシング，凝集度，ソフトウェアメトリクスについて述べる．3 章では本研究で提案するソフトウェアメトリクスに関して説明し，4 章でコードブロック間の結合度を用いたメソッド抽出支援手法に関して説明する．5 章では提案手法を実装したシステムに関して説明し，6 章では実装したシステムを用いた評価実験の結果を示す．8 章では関連研究を紹介する．9 章ではまとめと今後の課題について述べる．

2　背景

2.1　メソッド抽出リファクタリング

2.1.1　概要

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．Fowler はリファクタリングを必要とするソースコードの状態に応じて，さまざまなリファクタリングパターンを提案している [3]．本章では，それらのパターンのうちメソッド抽出リファクタリングについて述べる．メソッド抽出とは，メソッド内のコードの一部を新しいメソッドとして抽出する作業を意味する．

2.1.2　メソッド抽出手順

通常，メソッド抽出リファクタリングは次の手順で行われる [9, 21]．

手順 1 メソッドを抽出する必要があるメソッドを特定．

手順 2 新規メソッドとして抽出する範囲を識別．

手順 3 選択範囲をメソッドとして抽出．

手順 4 変更後のプログラムが外部的振る舞いを保っているか検証．

手順 1 は LOC(Lines Of Code) やサイクロマチック数 [8] などのソフトウェアメトリクスを用いてある程度自動化することができる [11, 12, 18]．

手順 2 はメソッド抽出手順の中でもとくに誤りが混入しやすく，必要な労力も大きい．具体的には，選択範囲と範囲外とのデータ依存や制御依存を調査しなければならない．また，メソッドを機能に基づいて分割するには，プログラムに込められた開発者の意図をある程度理解する必要があるが，開発者の意図を機械的に読み取ることは非常に難しい．

手順 3 は，自動化することが可能である．実際に Eclipse[2] などの統合開発環境やリファクタリングツール [16, 17] を用いることにより，自動的にソースコード修正を行うことができる．

手順 4 は抽出元のメソッドのテストケースを利用することによりある程度検証できる [3]．また，プログラム依存グラフを用いて 2 つのプログラムの挙動が等価であることを確認する手法も提案されている [6]．

2.1.3　メソッド抽出リファクタリングを必要とするメソッド

リファクタリングがどのようなとき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が要求される可能性を示す兆候 (不吉な匂い) を定義している [3].

以下では，メソッド抽出の必要性を示す既存の不吉な匂いに関して述べる．

長すぎるメソッド メソッドは長くなるほど可読性が低下する．長すぎるメソッドはメソッド抽出を行うことにより，短くすることが望ましい [3].

制御ロジックの複雑なメソッド プログラムの可読性を低下させる代表的な要因に複雑な制御ロジックがある．制御ロジックの複雑なメソッドは入力に応じて実行される処理が異なるため，可読性が低下する．このようなメソッドはメソッド抽出を行うことにより，制御ロジックを分解することが望ましい [3].

凝集度の低いメソッド 凝集度とはモジュール内の構成要素が協調している度合を示す [19]. 一般にモジュールは凝集度が高いほど保守性が向上する．これはメソッドに関しても同様である．このため，凝集度の低いメソッドはメソッド抽出により，互いに協調していない要素を分離することが望ましい [7].

2.1.4　適切なメソッド抽出

メソッド抽出は短いメソッドの形成を促すが，より適切なメソッド抽出を行うためには，短いだけでなく，次に述べる特徴を持つメソッドを形成することが望ましい．

機能に基づいて抽出されたメソッド メソッドは機能に基づいて抽出されることが望ましい．メソッド抽出を行うか否かの決定において，データ依存の有無や複雑度メトリクスなどの情報が有用であるが [5, 11, 18]，最終的な決定は機能に基づいて行うべきである．

凝集度の高いメソッド 一般にモジュールは機能に基づいて分割するだけでなく，さらに，機能とモジュールが 1 対 1 であることが望ましい．この度合は凝集度と呼ばれるメトリクスで表される．凝集度は高いほど適切に機能の局所化が行われていることを示し，そのモジュールの保守性や再利用性が高いことを意味する．これはメソッドに関しても同様である．

引数の少ないメソッド メソッドの引数は少ない方が望ましい [3]．引数が多いと各引数の役割が理解しづらくなる．さらに，各引数の役割の一貫性がなくなるため，そのメソッドを使用しづらくなる．また，呼び出し側のメソッドとの依存関係が強くなるため，呼び出し元のメソッドの変更の影響を受けやすくなる．

2.1.5　メソッド抽出の利点

メソッド抽出リファクタリングは，1つの長すぎるメソッドの一部を新しいメソッドとして抽出することにより，2つの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 [3].

- 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，メソッド抽出リファクタリング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．

このように，メソッド抽出リファクタリング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 [2, 12, 21].

2.2　プログラムスライシング

プログラムスライシング（以降，スライシング）とは，プログラム中の文間の依存関係に注目し，スライシング基準となる変数に依存関係のある文の集合であるプログラムスライスを抽出する技術である [20]. 通常，スライシングは制御依存関係やデータ依存関係に基づいてプログラム依存グラフを構築し，スラインシング基準に対応する頂点からのグラフ探索によって計算される．ここでデータ依存関係とは，ある変数 w が存在して，文 s における変数 w の定義が変数 w を参照している文 t に到達する場合の文 s と文 t の関係を意味する．制御依存関係とは，制御文 s と制御文 s 内に含まれる文 t の関係を意味する．

特定の機能 F に関連したプログラム要素をスライシング基準として指定することにより，ソースコード中から機能 F に関連したコード片を抽出することができる．このため，スライシングを用いて，メソッド抽出手順2をある程度自動化できると考えられる．しかし，多くの場合，データや制御は推移的に依存しているため，異なる機能に関連したスライシング基準を与えたとしても，得られる結果の多くは共通している [23]. 推移的な依存関係の列はデバッグなどの用途で有益な場合も多いが，スライシング基準に強く関連した局所的な範囲を抽出する場合には不要である．つまり，スライシングを用いて，メソッド抽出手順2を自動化するには適切なタイミングでグラフ探索を打ち切る必要がある．

2.3 凝集度

凝集度とはモジュール内の構成要素 (機能要素とデータ要素) が特定の機能を実現するため協調している度合を表す尺度である [19]．凝集度が高いほど各モジュールの役割が適切に分担されていることを示し，そのモジュールの堅牢性，再利用性，可読性などが向上する．

凝集度はその高さに応じて次のように分類される．

機能的凝集 モジュール内の全ての構成要素が単一の機能を実現するための処理に関連している状態．機能的凝集度を持つ下階層 (メソッドの場合，呼び出し元メソッドが上階層，呼び出されるメソッドが下階層) のモジュールを管理するだけのモジュールも機能的凝集度を持つ．

逐次的凝集 モジュール内のある部分の出力が，モジュール内の別の部分の入力となるよう状態．このときモジュール内には 2 つ以上の機能が含まれているが，それらの関連性は高く処理の順序に強い意味があるため 1 つのモジュールに収まっている．しかし，この状態のモジュールは肥大化し，可読性が悪化する傾向がある．

通信的凝集 同一の入力・出力に関連した機能が 1 つのモジュール内にまとめられている状態．ただし，それぞれの機能を実現するための処理に順序的な制約はない点で，逐次的凝集とは異なる．

手順的凝集 複数の処理が順序的な制約をもつため 1 つのモジュールにまとめられている状態．ただし，逐次的凝集と異なり，それぞれの処理が実現する機能の関連性は低い．

一時的凝集 モジュール内に互いに関連のない複数の機能がまとめられているが，それらの機能を実現するための処理が同一のタイミングで実行される状態．初期化処理をまとめたモジュールや例外処理をまとめたモジュールがこれに該当する．

論理的凝集 論理的に類似した複数の機能が 1 つのモジュールに集められている状態．この状態にあるモジュールは内部に持つ複数の機能を逐次的に実行するのではなく，外部のモジュールから選択された機能のみを実行する．

偶発的凝集 モジュールを構成する要素が無作為に集められており，互いに関連性を持たない状態．

凝集度は必ずしも前述の順に高いとは限らない．ただし，機能的凝集は最も望ましく，論理的凝集と偶発的凝集は他の凝集よりも凝集度が低い．また，逐次的凝集と通信的凝集は手順的凝集と一時的凝集よりも凝集度が高い．

2.4 ソフトウェアメトリクス

不吉な匂いのいくつかはソフトウェアメトリクスを用いて定量的に評価することができる．ソフトウェアメトリクスとは，ソフトウェアの品質評価や工数・保守コスト予測などに用いられる尺度である [13]．一般に，ソフトウェアメトリクスはクラスやメソッド，制御フローといったソフトウェアの構成要素に対して定義される．このため，いくつかのソフトウェアメトリクスはリファクタリングを必要とする可能性がある構成要素を定量的に評価するために利用できる．

以下では，メソッド抽出の必要性を示す不吉な匂いと，それを定量的に表すソフトウェアメトリクスに関して述べる．

LOC LOC はソースコードの行数を表す．このため，一定の値より LOC が高いメソッドは長すぎるメソッドであり，メソッド抽出リファクタリングの候補であるといえる．

サイクロマチック数 サイクロマチック数とは，MaCabe によって提案された循環的複雑度を表すメトリクスである [8]．プログラムの分岐点をノード，分岐点の間にあるコードブロックをエッジとし，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．つまり，サイクロマチック数が大きいほど，そのメソッドは制御ロジックが複雑であることを示す．経験的にサイクロマチック数は 10 以下に抑えることが望ましく [8]，10 を超えるメソッドはメソッド抽出の候補であるといわれている．

LCOM Chidamber と Kemerer はオブジェクト指向プログラムのクラスを対象とするソフトウェアメトリクスである CK メトリクスを提案している [1]．CK メトリクスは 6 つのメトリクスから構成され，その 1 つである LCOM(Lack Of Cohesion in Methods) はメソッドの凝集の欠如を評価する．メソッドの凝集性が高いほど，意味的に「閉じた」クラス設計がなされていることになり，良い設計であることを示す．

LCOM はクラスの属性をデータ要素，メソッドを機能要素と捉え，それらの関連性を計測することにより求められる．具体的には次式で表される．

$$LCOM = \begin{cases} P - Q & (if P > Q) \\ 0 & (otherwise) \end{cases}$$

P ： 属性を共有していないメソッドのペアの数
Q ： 1 つ以上属性を共有しているメソッドのペアの数

LCOM は凝集の欠如を表すため低いほど望ましい．

LCOM* Henderson はより正確にメソッドの凝集の欠如を表すために LCOM*と呼ばれる凝集度メトリクスを提案している [4]．メソッドの凝集の欠如は Chidamber と Kemerer によって定義された LCOM では正確に表せない場合があることが知られている．

LCOM*ではモジュールの機能が適切に局所化されていれば，モジュール内の特定のデータ要素にアクセスするモジュール内の機能要素の数は増加すると定義している．具体的には，LCOM と同様にクラスの属性をデータ要素，メソッドを機能要素と捉えたとき，次式で表される．

$$LCOM* = \frac{\frac{1}{a}\sum_{j}^{a}\mu(A_j) - m}{1-m}$$

$$\begin{array}{rcl} A_j & : & \text{着目しているクラスの } j \text{ 番目の属性} \\ a & : & \text{着目しているクラスの属性の数} \\ m & : & \text{着目しているクラスのメソッドの数} \\ \mu(A_j) & : & \text{属性 } A_j \text{にアクセスするメソッドの数} \end{array}$$

LCOM*は LCOM と同様に凝集の欠如を表すため低いほど望ましい．

スライスベースの凝集度メトリクス Weiser はメソッドや関数 (以降，関数はメソッドと同一のものとして扱う) の凝集度を評価するためにプログラムスライスを利用した 5 つの凝集度メトリクスを提案した [20]．また，これらの凝集度メトリクスは Ott と Bieman によって定式化され，実験により 5 つうち Tightness，Overlap，Coverage と呼ばれる 3 つのメトリクスの有用性が高いことが示されている [14]．Tightness はメソッド内の文の数と全スライスに含まれる文の数の比率を，Overlap は各スライスに含まれる文の数と全スライスに含まれる文の数の平均比率を，Coverage はメソッド内の文の数と各スライスに含まれる文の数の平均比率を表す．具体的には，計測対象のメソッドを M としたとき，次式で表される．

$$\begin{array}{rcl} Tightness(M) & = & \frac{|SL_{int}|}{length(M)} \\ Overlap(M) & = & \frac{1}{V_O}\sum_{x\in V_O}\frac{|SL_{int}|}{||SL_x|} \\ Coverage(M) & = & \frac{1}{V_O}\sum_{x\in V_O}\frac{|SL_x|}{length(M)} \end{array}$$

$$\begin{array}{rcl} length(M) & : & M \text{ の文の数} \\ V_O & : & M \text{ の出力変数 (返り値など) の集合} \\ SL_x & : & \text{変数 } x \in V_O \text{を基点としたスライス} \\ SL_{int} & : & \bigcap_{x\in V_O} SL_x \\ & & (M \text{ 内の全スライスの共通部分}) \end{array}$$

Meyers と Binkley は上記の定義を用いて，複数のドメイン，かつ，複数のバージョン

のオープンソースソフトウェアを対象に実証的な実験を行っている．彼らの行った実験により，Tightness は Overlap と Coverage の両方と相関があるこが示されている．

スライスベースの凝集度メトリクスは 7 つの凝集度レベルのうち下位 4 つを凝集度の低いモジュール (メソッド抽出の候補) として識別できる．しかり，機能的凝集より低い逐次的・通信的凝集にあるモジュールを識別することができない場合が多い．これは出力変数 (返り値やメソッド内で変更される属性など) のみに着目してメトリクスが計測されるためである．

3　ソフトウェアメトリクス: COB

オブジェクト指向には，処理 (機能要素) および処理に必要なデータ (データ要素) を 1 つにまとめるという重要な考え方がある．同様に，機能要素が必要としないデータ要素は互いに分離しておくことが望ましい．この考えに基づき構成されたモジュールは機能要素とデータ要素が互いに協調しており，凝集度が高くなる．クラスの場合，その度合を表すメトリクスとして LCOM が提案されている [1, 4]．LCOM は複数の定義が提案されているが，本質的には属性をデータ要素，メソッドを機能要素とみなし，その協調度合を表す.

本研究では，メソッドの構成要素の協調度合を示すため，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し，ソフトウェアメトリクス COB(Cohesion Of Blocks) を次のように定義する．b はメソッド内のコードブロックの数，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の数を示す.

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

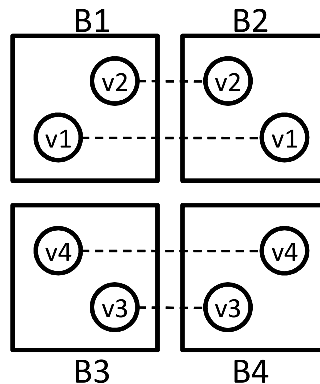
COB は LCOM*[4] と同様に，特定のデータ要素と協調する機能要素の割合の平均に着目している．COB の値の最小値は 0，最大値は 1 であり，値が大きいほど構成要素の協調度合が高いことを示す.

図 1(a) の $sample$ メソッドコードを用いて COB 計測の例を示す．$sample$ メソッド内の BLOCK1-4 はコードブロックを示す．また，図 1(b) は $sample$ メソッドの構成要素であるコードブロックと使用されている変数の協調関係を抽象化した図である．図 1(b) において四角はコードブロックを，四角内の円はそのコードブロック内で使用されている変数を，点線は 2 つのコードブロックが変数を介して協調していることを意味する．図 1(b) からわかるように，$sample$ メソッドは BLOCK1 と BLOCK2 は変数 $v1$,$v2$ を介して協調している．同様に BLOCK3 と BLOCK4 も協調している．一方，BLOCK1，2 で構成されるコード片と BLOCK3，4 で構成されるコード片は協調していない．このとき COB の値は 0.5 となる．変数 $v4$ の使用を変数 $v1$ の使用に置き換えたとき，図 1(c) のように全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 0.66 に上昇する．さらに，変数 $v3$ と使用を変数 $v2$ の使用に置き換えると全変数が全コードブロックで使用され COB は最大値 1 となる.

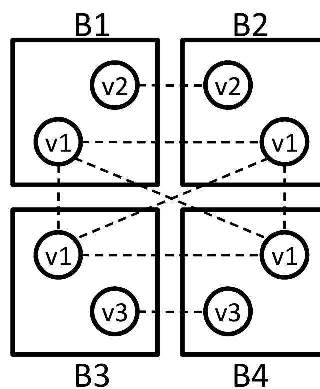
ここで，互いに呼び出し関係にある 2 つのメソッドについて考える．一方のメソッド内のローカル変数は基本的にもう一方のメソッドからはアクセスできない．引数を利用してアクセスすることは可能であるが，2 つのメソッドが適切に切り分けられているとき引数の数は少なくなる傾向がある．このため，適切に切り分けられた 2 つのメソッドを 1 つにまとめたとき，メソッド内の要素の協調状況は図 1(b) と似たものになり，COB は低くなる．このこ

```
void sample() {
   int v1, v2, v3, v4;
   ..........
   BLOCK1 {
      v1 = v1 + v2;
   }
   BLOCK2 {
      v2 = v1++;
   }
   BLOCK3 {
      v3 = v3 + v4;
   }
   BLOCK4 {
      v4 = v3++;
   }
}
```

(a) sample メソッド



(b) sample メソッドの抽象図



(c) sample メソッドの抽象図 (変数 v4 を変数 v1 に置換)

とから，COB の低いメソッドはメソッド抽出を行い，2 つに分割するべきメソッドの候補であるといえる．

既存のメトリクスと比べ，COB は次の利点を持つ．

- メソッド内の要素の協調度合を表すため，LOC やサイクロマチック数より機能に基づいた評価が行える．
- LOC やサイクロマチック数と異なり適切にメソッド抽出した場合，元のメソッドと新規メソッドのメトリクス値が高くなる．例えば，図 1(b) において BLOCK1 と BLOCK3 をメソッド抽出しても COB は変化しないが，BLOCK1 と BLOCK2 をメソッド抽出した場合は上昇する．
- 出力変数だけでなく全ての変数に着目するため，逐次的・通信的凝集の方が機能的凝集よりもメトリクス値が低くなる傾向がある．

4 コードブロック間の結合度を用いたメソッド抽出範囲の識別

本節では新規メソッドとして抽出する範囲 (以降，メソッド抽出範囲) の識別作業を支援するための手法について述べる．本手法は次の 3 つの点に着目してメソッド抽出範囲の識別を支援する．

- メソッド内の処理は主に演算やメソッド呼び出しにより実現される．演算は多くの場合，変数の値をもとに行われ，その結果も変数に格納される．また，オブジェクト指向プログラミング言語の場合，メソッド呼び出しも変数を介することが多い．丸山はこのことに着目し，メソッドの機能は返り値となる変数の値を決定する計算であると考え，返り値を基点としたプログラムスライスを求めることによりメソッド抽出範囲を識別する手法を提案している [21]．本手法も丸山の手法と同様に変数に着目して，機能に基づいたメソッド抽出を試みる．ただし，メソッド内の処理で値が決定される変数は必ずしも 1 つではなく，変数の値を決定するために必要な変数も存在するため，本手法では複数の変数に着目する．

- 本手法では 2 つのコード片が同一の機能に関連しているか否かを，データ依存ではなく共通して使用している変数に着目して判別する．一般に，1 つの変数は 1 つの役割を持つようにプログラムを記述するべきである．このため，データ依存があってもデータを格納する変数が異なれば，そのデータの役割が変わっているため機能の分割点が存在する可能性は高い．

- 本手法では 2 つのコード片で共通して使用しているデータの絶対量ではなく，相対量に着目する．一般に，機能は複雑であるほど，扱うデータの量が増える傾向にある．このため，2 つのコード片で共通して利用しているデータの絶対量が多い場合でも，実現しようとする機能が複雑であれば，共通して利用しているデータの相対量は少ない可能性が高い．複雑な機能は機能分割することにより簡単化することが望ましいため，共通して利用するデータの絶対量が多い場合でも，相対量が少なければ機能の分割点として注目する価値は高い．

  これは前章で提案した COB の定義にも矛盾しない．例として 2 つのコード片 C_1 と C_2 に関して考える．コード片 C_1 で使用されている変数の集合を V_1，コード片 C_2 で使

用されている変数の集合を V_2 としたとき，次の式が成り立つ．

$$
\begin{aligned}
COB &= \frac{1}{2}\frac{1}{|V_1 \bigcup V_2|}(|V_1| + |V_2|) \\
&= \frac{1}{2}\frac{|V_1 \bigcup V_2| + |V_1 \bigcap V_2|}{|V_1 \bigcup V_2|} \\
&= \frac{1}{2} + \frac{1}{2}\frac{|V_1 \bigcap V_2|}{|V_1 \bigcup V_2|}
\end{aligned}
$$

このように COB の定義に着目したとき，2 つのコード片 C_1 と C_2 が協調している度合は共通しているデータの相対量 $\frac{|V_1 \bigcap V_2|}{|V_1 \bigcup V_2|}$ に依存する．LCOM*の定義においても同様のことが成り立つ．LCOM は共通しているデータの絶対量に着目しているが，2.4 で述べたように，凝集度をうまく表せない場合が存在することがわかっている．

これらのことから，共通して利用しているデータの相対量に着目することは凝集度の高いメソッドの構築，ひいては，機能に基づいたメソッド分割につながると考えられる．

以降，本章では，変数を用いた機能の抽象化について述べ，次に本手法の詳細を各ステップごとに述べる．

4.1 変数を用いた機能の抽象化

既に述べたように，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．ただし，for 文の初期化部で宣言されるカウンタ変数などは集合の要素に含まない．

$$F(M) = \{v_1, v_2, v_3 \cdots v_{n-1}, v_n\} \tag{1}$$

(v_i : M 内で使用されている変数)

同様に考えると，メソッド M 内のコードブロック B の機能 $F(B)$ はコードブロック 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 $F(B) \subseteq F(M)$ 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．コードブロック 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．

本手法では，このことを踏まえ同一の機能に属するコードブロック群を識別する．

4.2 コードブロック間の関係

2 つのコードブロックの関係を表す概念として，兄弟関係と親子関係を定義する．2 つの関係に関して図 1，2 を用いて説明する．図 2 は図 1 のコードのコードブロック構造を木構造で表現している．木構造におけるノードは対応するコードブロックを意味している．

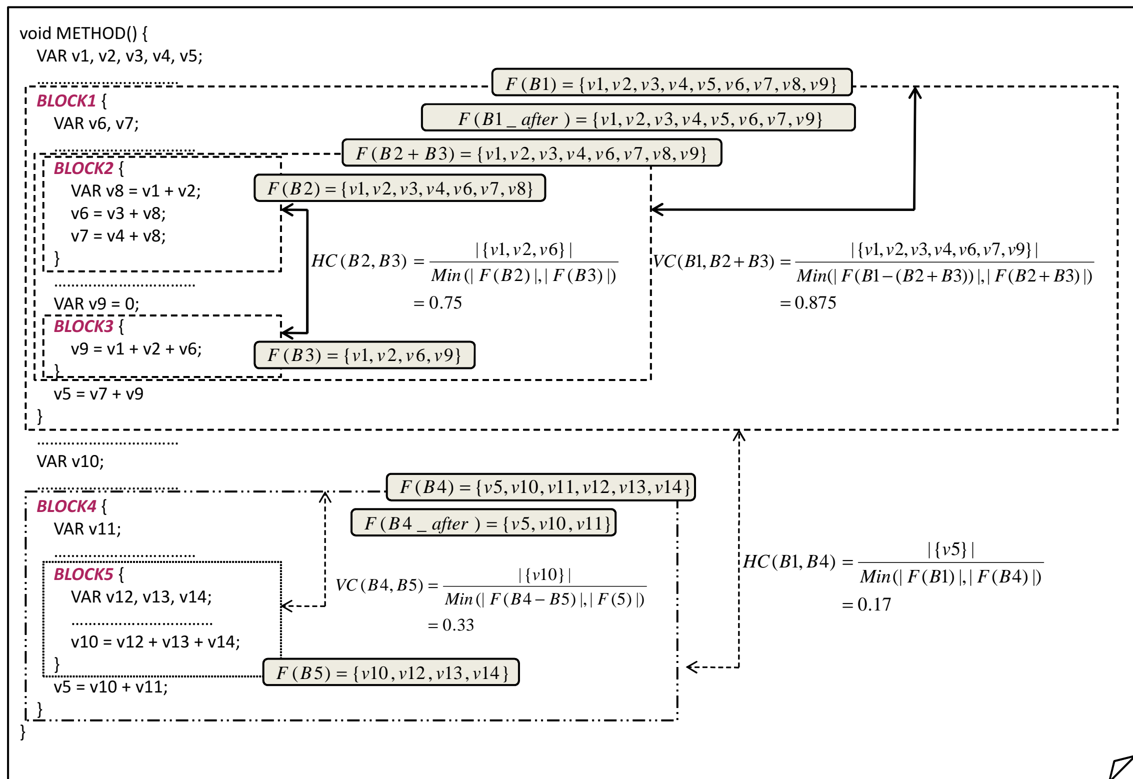


図 1: コードブロック間の結合度を用いた機能分割

兄弟関係 木構造において，兄弟ノードとなる 2 つのコードブロックの関係を兄弟関係と定義する (点線の両矢印で表示)．つまり，BLOCK1 と BLOCK4，BLOCK2 と BLOCK3 は互いに兄弟関係を持つ．

親子関係 木構造において，親子ノードとなる 2 つのコードブロック間の関係を親子関係と定義する (実線の両矢印で表示)．つまり，BLOCK1 と BLOCK2，BLOCK1 と BLOCK3，BLOCK4 と BLOCK5 は互いに親子関係にある．

4.3 コードブロック間の結合

兄弟関係もしくは親子関係にある 2 つのコードブロックが同一の機能に属するか否かを，次の 2 つの結合度を用いて識別する．結合度が大きいとき 2 つのコードブロックが同一の機能に属すると判断する．

兄弟ブロック結合度 兄弟関係を持つコードブロック間の結合度を表す．あるコードブロック B_1 の機能を $F(B_1)$，もう一方のコードブロック B_2 の機能を $F(B_2)$ とすると，2つのコードブロックの兄弟ブロック結合度 $SBC(B_1, B_2)$ 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．

$$SBC(B_1, B_2) = \frac{|F(B_1) \cap F(B_2)|}{Min(|F(B_1)|, |F(B_2)|)} \quad (2)$$

$(Min(A, B):$ A と B のうち小さい方の値$)$

$SBC(B_1, B_2)$ 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 $SBC(B_1, B_2)$ 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，$SBC(B_1, B_2)$ の値が閾値より大きければ，一方のコードブロックの機能はもう一方のコードブロックの機能の一部，もしくは，類似した機能であると判断できる．

兄弟ブロック結合度の閾値を 0.5 としたときの例を図 1 に示す．コードブロック 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．同様にコードブロック 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，兄弟ブロック結合度 $SBC(B_2, B3)$ 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 の兄弟ブロック結合度は 0.17 であり，閾値より低いため 2 つのコードブロックは異なる機能に属すると判断する．

親子ブロック結合度 親子関係を持つコードブロック間の結合度を示す．ここで，あるコードブロック B_{in} の機能を $F(B_{in})$，B_{in} の外側のコードブロック (木構造上の親ノード)B_{out} の機能を $F(B_{out})$ とする．また，コードブロック B_{in} 内で宣言している変数群を $DV(B_{in})$ とする．このとき，コードブロック B_{in} がメソッドとして抽出されると仮定すると，抽出後の外側のコードブロック B_{out_after} にとって，内側のコードブロック 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．

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

抽出元のメソッドに残るコードブロックの機能 $F(B_{out_after})$ と新規メソッドとして抽出されたコードブロックの機能 $F(B_{in})$ 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，内側のコードブロック B_{in} のみでなく，外側のコードブロック B_{out} ごと抽出す

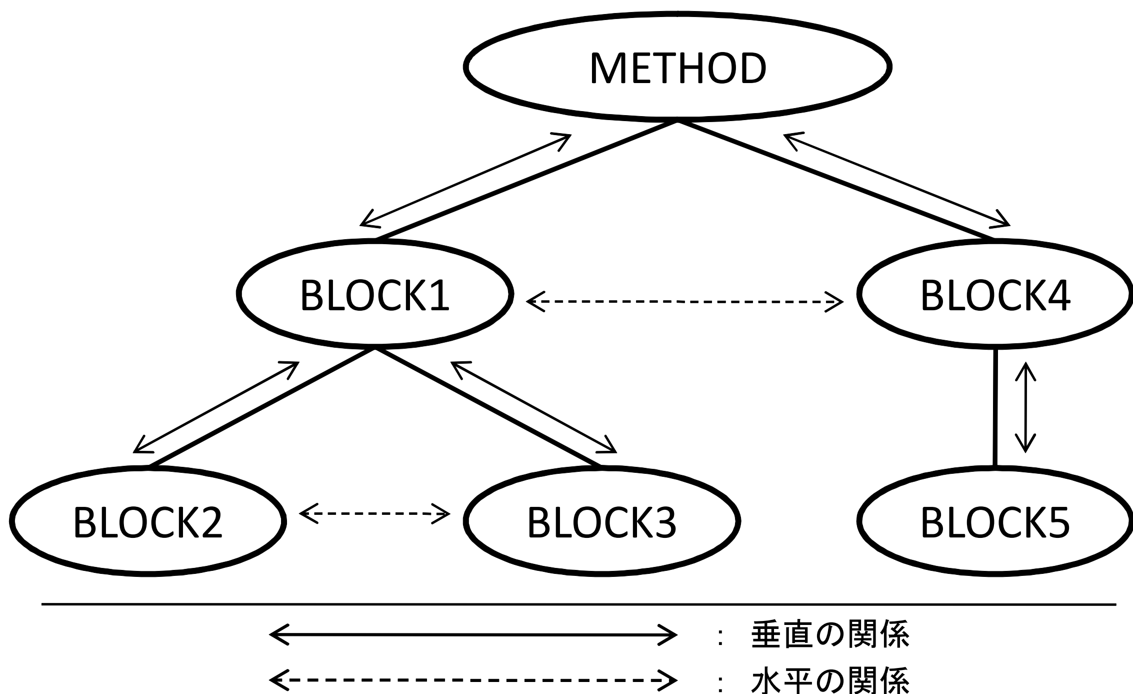


図 2: コードブロック間の関係

る必要がある．以上のことを考慮し，2 つのコードブロック B_{in}，B_{out} の親子ブロック結合度 $PBC(B_{in}, B_{out})$ は抽出元のメソッドに残るコードブロックの機能 $F(B_{out_after})$ と新規メソッドとして抽出されたコードブロックの機能 $F(B_{in})$ の類似度で表す．具体的には次の式で表される．

$$PBC(B_{in}, B_{out}) = \frac{|F(B_{out_after}) \cap F(B_{in})|}{Min(|F(B_{out_after})|, |F(B_{in})|)} \tag{3}$$

$PBC(B_{in}, B_{out})$ が大きいとき，兄弟ブロック結合が大きいときと同様の理由で，内側のコードブロックの機能は外側のコードブロックの機能の一部，もしくは，類似した機能であると判断できる．

親子ブロック結合度は兄弟関係を持つ複数のコードブロック群とそれらの外側のコードブロックに対しても計測することができる．この場合，内側のコードブロック群の機能は各コードブロックの機能を表す集合の和集合で表される．

親子ブロック結合度の閾値を 0.5 としたときの例を図 1 に示す．コードブロック 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 $v12$，$v13$，$v14$ 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，親子ブロック結合度 $PBC(B_4, B_5)$ 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 の親子ブロック結合度は 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．

4.4 同一機能に属するコードブロック群の識別

あるコードブロックと同一の機能に属すると考えられるコードブロック群をコードブロック間の結合を用いて，次のアルゴリズムで識別する．

アルゴリズム

```
1. RESULT = NULL
2. B = 基点となるコードブロック
3. SCB = B と結合している兄弟コードブロック群 (B を含む)
4. NSCB = B と結合していない兄弟コードブロック群
5. PB = B の親ブロック
6. if (NSCB が空でない)
7.   if (PBC(OB,SCB) > PBC(OB,NSCB))
8.     NSCB を新規メソッドとして抽出
9.   else
10.    SCB を新規メソッドとして抽出
11.    EXIT
12.  end if
13. end if
14. if(OB と SCB が親子ブロック結合している)
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.  SCB を新規メソッドとして抽出
21. end if
```

アルゴリズムを図 1 のコードに適用し，BLOCK2 と同一の機能に属するコードブロック群を識別する例を示す．兄弟ブロック結合と親子ブロック結合の閾値はどちらも 0.5 とする．このとき BLOCK2 は BLOCK3 と兄弟ブロック結合しており (行 3)，BLOCK1 内に BLOCK2 と兄弟ブロック結合していないコードブロックは存在しない (行 4,5,6)．BLOCK2 と BLOCK3 を含むコード片は BLOCK1 と親子ブロック結合している (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 と兄弟ブロック結合しておらず (行 4,6)，外側のブロックであるメソッド本体とは BLOCK1 の方が強く親子ブロック結合している (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親子ブロック結合している (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．

4.5　同一の機能に属する周囲のコードの識別

4.4 のアルゴリズムでメソッドとして抽出すべきコードブロック群が識別される．しかし，より精度の高い機能分割を実現するためには，それぞれのコードブロックの前後のコード片も，コードブロック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．同一の機能に属するコード片の識別にはプログラムスライシングが利用できる [23].

本手法では，手続き間解析を行わない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コードブロックの外側で宣言されている変数 (以降，外部変数と呼ぶ) のうち，コードブロックと関連が強い変数を指定する．ここでコードブロック B と変数 v の関連とは，コードブロック B の機能の実現に変数 v がどの程度貢献しているかを意味する．関連の強さは次の 2 つのヒューリスティックを用いて判断する．

$\mathbf{RefRate(v,B)}$ v の全参照のうち B 内での参照が占める割合．この値は v の参照が B の外部より内部で多いとき 0.5 を超える．このため，$RefRate(v,B)$ が 0.5 以上のとき B と v は強く関連していると判断する．

$\mathbf{isAssigned(v,B)}$ B 内で v が代入されているか否か．メソッドの機能はその内部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．同様に，B の内部で v に値が代入されている場合，B の機能は v の値を決定することであると考えられため，B と v は強く関連していると判断する．ただし，変数の初期化処理は代入処理に含まない．

以上のヒューリスティックを用いて，コードブロック B と関連の強い外部変数の集合 $ROV(B)$ を次のように定義する．

$$
\begin{aligned}
v \;\in\; & ROV(B): \\
& (RefRate(v,B) > 0.5) \vee isAssigned(v,B)
\end{aligned}
\tag{4}
$$

$ROV(B)$ の要素 rov を基点としたスライス範囲 $SL(rov)$ は次のように定義する．

スライス始点: 変数 rov の宣言位置

スライス終点: 変数 rov の最後の参照・代入位置

SL(rov): 始点から終点までのフォワードスライス

$SL(rov)$ を計算することにより，変数 rov と依存関係があるコードが識別される．このとき，変数 rov はコードブロック B と関連が強いため，変数 rov と依存関係があるコード片 $SL(rov)$ もコードブロック B と関連が強いと考えられる．ゆえに，コードブロック B と同一の機能に属する範囲 $ExtractedRange(B)$ は次のように定義できる．

$$ExtractedRange(B) = B \cup \bigcup_{rov \in ROV(B)} SL(rov) \tag{5}$$

コード片 $ExtractedRange(B)$ を新規メソッドとして抽出することにより，機能ごとの分割を実現することができる．また，抽出元のメソッドから $ROV(B)$ は不可視になるため，保守性や可読性などの向上が見込める．

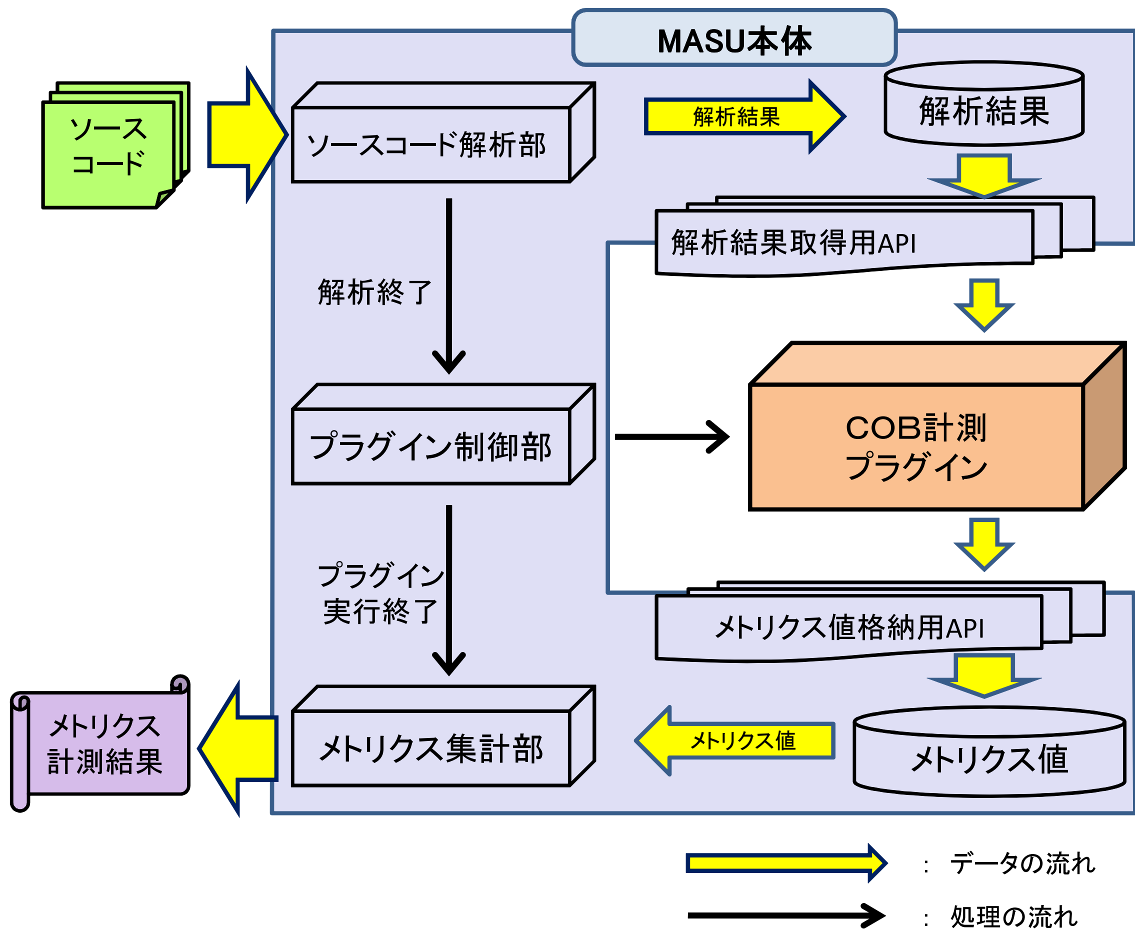


図 3: COB 計測プラグインと MASU のシステム構成

5　実装

本節では 3 章で提案したソフトウェアメトリクス COB を計測するシステムと，4 章で説明したメソッド抽出範囲識別手法を実装したシステムに関して説明する．いずれのシステムも開発言語は Java である．

5.1　COB 計測プラグイン

ソフトウェアメトリクス COB を計測するシステムはメトリクス計測プラグイン開発基盤 MASU[22] を用いて MASU のプラグインとして実装した．

システムの構成図を図 3 に示す．システムの処理の流れは次の通りである．

(1) 計測対象のソフトウェアを MASU の本体に入力

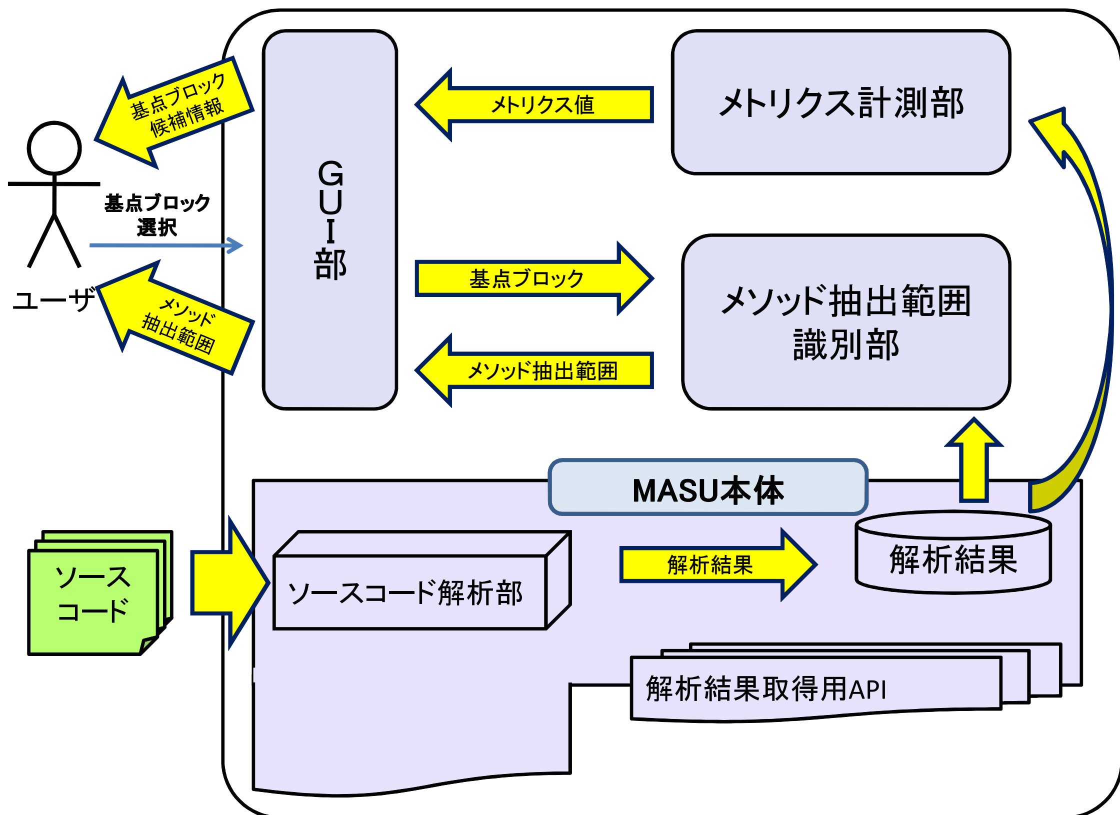


図 4: メソッド抽出範囲識別ツールのシステム構成

(2) COB 計測プラグインを MASU の本体に登録

(3) MASU の本体はソースコードを解析して，COB 計測に必要な情報を抽出

(4) COB 計測プラグインは MASU の本体から提供される情報を用いて COB 計測ロジックを実行

(5) 計測された COB の値を MASU の本体に提供

(6) MASU の本体は計測された COB の値を集計し，メソッドのシグネチャと対応する COB の値を CSV 形式でファイルに出力

5.2 メソッド抽出範囲識別ツール

4 章で説明したメソッド抽出範囲識別手法を実装したツール (以降，本ツール) に関して説明する．本ツールの概要を図 4 に示す．

本ツールはソースコード解析部，メトリクス計測部，メソッド抽出範囲識別部，GUI 部で構成される．

ソースコード解析部 メトリクス計測部やメソッド抽出範囲識別部で必要な情報をソースコードから抽出する．実際のソースコード解析は MASU の本体のソースコード解析部が行い，その解析結果を利用している．

メトリクス計測部 メソッド抽出範囲識別の基点となるコードブロックを選択する際の参考情報として，メソッドおよびコードブロック単位で LOC やサイクロマチック数などのソフトウェアメトリクスを計測する．コードブロックの一覧と計測されたメトリクス値は GUI 部に表示される．

メソッド抽出範囲識別部 4 章で説明した手法を実装している．GUI 部に表示されたコードブロックから基点となるコードブロックを選択したとき，指定したコードブロックと同一の機能に含まれる範囲を識別し，GUI 部に表示する．

GUI 部 各部の解析結果を視覚化する．ユーザは表示された情報をもとに，メソッド抽出を行うか否かを決定する．

本ツールを使用した場合，ユーザは次の手順でリファクタリングを行う．

(1) 入力としてソースコードを与える．

(2) メトリクス計測部で計測された情報をもとにメソッド抽出の基点となるコードブロックを選択する．

(3) ツールにより特定されたメソッド抽出対象範囲を確認し，メソッド抽出を行うか否か決定する．

(4) Eclipse などの既存ツールを使用し，メソッド抽出を行う．

6 適用実験

本節では 5 章で紹介したシステムを既存のソフトウェアに適用した事例について述べる．システムの実行環境は以下の通りである．

- CPU: Intel(R) Pentium(R)4 CPU 3.00GHz
- RAM: 2.00GB
- OS: Mictrosoft Windows XP Professional Version2002 Service Pack2

6.1 COB を用いたメソッド抽出候補の検出

既存のソフトウェア apache-ant-1.7.0(表 1 参照) に対して COB，LOC(空白・コメントを含む)，サイクロマチック数を計測し，それぞれのメトリクスごとに，メトリクス値の大きさでソートし，上位 20 メソッドをメソッド抽出の候補として検出した．

計測の結果，COB でソートした場合は 0.066 以下，LOC の場合は 146 以上，サイクロマチック数の場合は 22 以上のメトリクス値をもつメソッドがメソッド抽出の候補として検出された．検出されたメソッドのうち COB でソートした場合のみ検出された 6 つのメソッドを表 2 に示す．6 つのメソッドをメソッド名やコードコメントをもとに調査した結果，*JUnitTestRunner#main* メソッド以外のメソッドはメソッド抽出の対象として適当であると判断できた．*JUnitTestRunner#main* メソッドに関してはコードコメントが存在しなかったため評価を控えた．

図 5 にメソッド抽出候補の例としてメソッド *Concat#validate* の概略を示す．このメソッドは 6-37 行の範囲に含まれるコードブロック群内では 10 個の変数が，38-89 行の範囲に含まれるコードブロック群では 14 個の変数が使用されていたが，6-37 行と 38-89 行の両方の範囲内で使用されている変数は 2 個であった．つまり，片方の範囲で使用されている変数のほとんどがもう片方では使用されておらず，互いに協調していなかった．このため，COB が

表 1: Apache Ant の概要

ソフトウェア名	Apache Ant
種別	ビルドツール
バージョン	1.7.0
総行数	198,718
メソッド数	9,937

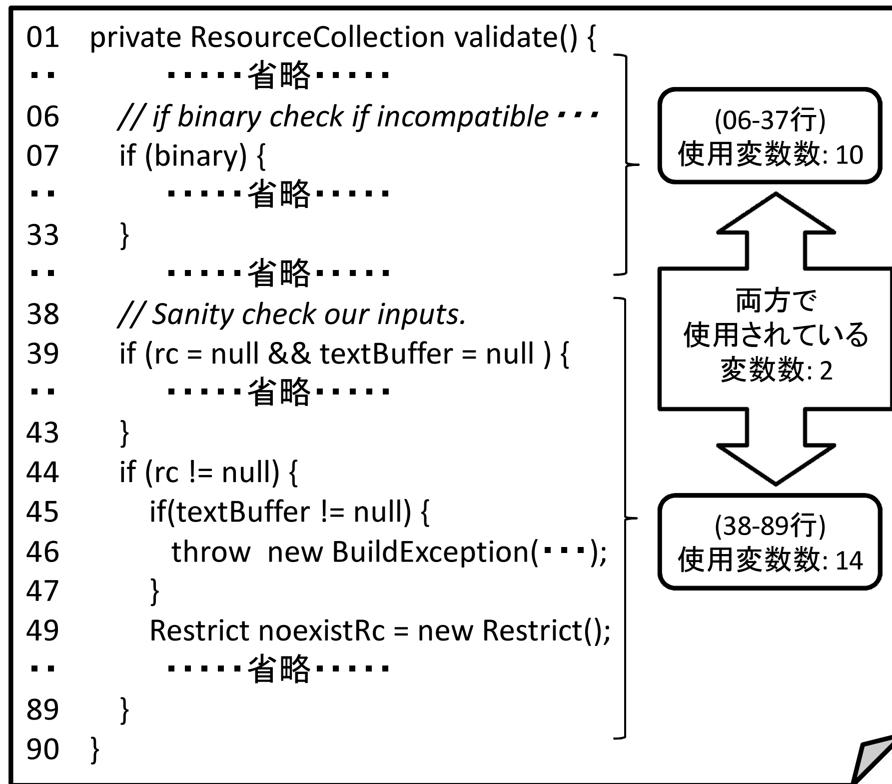


図 5: COB を用いて検出されたメソッド抽出候補の例

減少し，メソッド抽出の候補として検出された．また，このメソッドは 6 行と 38 行のコードコメントから，6-37 行のコード片はバイナリデータとして入力の正当性検証を，38-89 行のコード片はソースコードとして入力の正当性検証を行っていることがわかり，38 行を境に 2 つのメソッドに分割することが適当と判断できた．

表 2: COB を用いた場合のみメソッド抽出の候補となるメソッド

メソッド名	COB	LOC	サイクロマチック数
DirectoryScanner#scan	0.066	64	12
ModifiedSelector#configure	0.065	115	18
Parallel#spinThreads	0.065	136	17
JUnitTestRunner#main	0.065	106	19
Concat#validate	0.060	89	20
MacroDef#sameOrSiminlar	0.055	55	16

スライスベースの凝集度メトリクスはCOBとは異なる要素 (プラグラムスライス) の協調度合に着目しており，メソッド抽出の候補として検出されるメソッドもCOBを用いて検出されるメソッドとは特徴が異なる．また，LOCのような行ベースのソフトウェアメトリクスとは相関がないことが定量的に示されている [10]．このため，今回の適用事例では比較対象には含んでいない．

6.2 SBC/PBCを用いたメソッド抽出範囲の識別事例

5.2節で紹介したシステムを利用して，学生の書いたプログラムに含まれる既存のメソッドから，新規メソッドとして抽出すべき範囲を識別する例を図6に示す．この例では，基点となるコードブロックとして図6の $if1$ を指定した．図6のソースコード上で強調されているコード片は，基点となるブロック文 $if1$ と同一の機能に含まれると見なされたコード片である．また，表3は図6のソースコードにおける各コードブロックの機能を抽象化した際に，構成要素となる変数を表している．

4.4節のアルゴリズムにもとづいて $if1$ と同一の機能に含まれるコードブロック群を識別する過程を説明する．この例では，各コードブロックの結合の閾値は0.5とした．まず，$if1$ と $for4$ の親子ブロック結合 $PBC(if1, for4)$ を計測する．表3と結合度の導出式3より，$PBC(if1, for4)$ は1であり，閾値以上であるので $for4$ がメソッド抽出対象範囲に含まれる．実際に，$for4$ 内のコードは $if1$ が全てを占めており，$if1$ と $for4$ は同一の機能に属すると考えられる．次に $for4$ と $for3$ の兄弟ブロック結合 $SBC(for4, for3)$ を計測する．$for4$ と $for3$ の共通して使用している変数は $coverRangeArray$ のみであるが，$for3$ の構成要素数が2であるため，$SBC(for4, for3)$ は閾値以上となり，$for3$ もメソッド抽出範囲に含ま

表3: 図6のソースコード上のコードブロックの機能抽象化

コードブロック	抽象化された機能を構成する要素 (変数)
$if1$	$codeRangeArray, coverSize, previousRange$
$for4$($if1$ 抽出後)	$codeRangeArray, coverSize, previousRange$
$for4$	$codeRangeArray, coverSize, previousRange$
$for3$	$codeRangeVector, codeRangeArray$
$for2$	$cloneData, codeRangeVector, fileIndexRange2,$ $range, clonePair, left, right$
$for1$(強調部分抽出後)	$cloneData, codeRangeVector, finelIndexRange1,$ $fileIndexRange2, range, clonePair, left, right$

れる．実際に $for3$ では $for4$ で使用する変数 $codeRangeArray$ の初期化処理を行っており，$for4$ の機能の一部であると考えられる．同様に $for2$ と $for4$ の兄弟ブロック結合，コードブロック群 $for2+for3$ と $for1$ の親子ブロック結合を計測した結果，結合度は閾値より小さく，$for1$ と $for2$ はメソッド抽出範囲に含まれないと判定された．

上記の処理により識別された範囲は特定の機能に関連しており，メソッド抽出対象として妥当であると判断することができた．妥当性は次の 3 点に基いて評価した．

- コードを読み，著者の主観に基づき評価した．
- コード上のコメントから客観的に評価した．
- 識別されたコード片と重複したコード片が 3 つのクラスで確認され，また，それらの前後の文脈が異なっていたことから再利用可能な機能的単位であると判断した．

FileOrder.java

```
//-----------------------------------------------------------------
private double calculateGroupCoverage(int group1, int group2)
{
    Range fileIndexRange1 = cloneData.getFileIndexRange(group1);
    Range fileIndexRange2 = cloneData.getFileIndexRange(group2);

    // group1側の各ファイルのサイズを全て足す
    int group1TotalSize = 0;
    for (int fileIndex1 = fileIndexRange1.from; fileIndex1<= fileIndexRange1.to; fileIndex1++)
        group1TotalSize += cloneData.getFileDescAt(fileIndex1).tokenCount;

    double groupCoverage = 0.0;

    // group1側の各ファイルごとにカバレッジをだす
    for (int fileIndex1 = fileIndexRange1.from; fileIndex1<= fileIndexRange1.to; fileIndex1++)
    {
        // fileIndex1に関連したコードレンジを収集
        Vector codeRangeVector = new Vector();
        for (int fileIndex2 = fileIndexRange2.from; fileIndex2<= fileIndexRange2.to; fileIndex2++)
        {
            Range range = cloneData.getClonePairIndexRange(fileIndex1, fileIndex2);
            if (range == CloneData.EMPTY_RANGE) continue;

            for(int i = range.from; i <= range.to; i++)
            {
                ClonePair clonePair = cloneData.getClonePairAt(i);
                CodeFragment left  = clonePair.left;
                CodeFragment right = clonePair.right;

                if (fileIndex1 == left.file)
                    codeRangeVector.add(new Range(left.fromTokenIndex, left.toTokenIndex));
                else if (fileIndex1 == right.file)
                    codeRangeVector.add(new Range(right.fromTokenIndex, right.toTokenIndex));
                else System.out.println("internal error: calculateGroupCoverages");
            }
        }

        if (codeRangeVector.size() == 0) continue;

        // 配列に変換
        Range[] codeRangeArray = new Range[codeRangeVector.size()];
        for(int i = 0; i < codeRangeArray.length; i++)
            codeRangeArray[i] = (Range)codeRangeVector.get(i);

        // 前の範囲の方から順に並び替える
        Arrays.sort(codeRangeArray);

        // 何行カバーできているか算出
        int   coverSize = 0;
        Range previousRange = codeRangeArray[0];
        for (int i = 1; i < codeRangeArray.length; i++)
        {
            // 前の範囲と連続(もしくはオーバーラップ)している場合
            if (previousRange.to >= codeRangeArray[i].from)
            {
                // 範囲が拡大される場合
                if (previousRange.to < codeRangeArray[i].to)
                    previousRange.to = codeRangeArray[i].to;
            }
            // 前の範囲と非連続
            else
            {
                coverSize = coverSize + (previousRange.to - previousRange.from + 1);
                previousRange = codeRangeArray[i];
            }
        }
        coverSize = coverSize + (previousRange.to - previousRange.from + 1);

        double coverage = (double)coverSize / (double)(cloneData.getFileDescAt(fileIndex1).toke

        // group1におけるfileIndex1の重みを出す
        double rate = cloneData.getFileDescAt(fileIndex1).tokenCount/(double)group1TotalSize;

        groupCoverage += coverage * rate;
    }

    groupCoveragesBackUp[cloneData.getGroupIndex(group1)][cloneData.getGroupIndex(group2
    return groupCoverage;
}
```

for1 for2 for3 for4 if1

図 6: メソッド抽出対象範囲の識別例

7 考察

COB とスライスベースの凝集度メトリクスは異なる要素に着目して協調度合を評価している．具体的には，COB はメソッドの構成要素として変数とコードブロックに着目している．一方，スライスベースの凝集度メトリクスはメソッドの構成要素としてプログラムスライスに着目している．このため，両方のメトリクスを用いることにより，機能に基づいたメソッド抽出候補の検出を多角的に行うことができると考えられる．

本手法では変数の使用に着目することにより，コードブロック間の結合度を定義している．その際，使用している変数の重要度はすべて等価として扱っている. しかし，プログラムにおいて各変数の重要度は一律ではない. 例えば，4.5 節で説明したように，参照回数の多い変数や代入が行われている変数のほうが重要度は高いと考えられる. また，値型の変数より配列型や参照型の変数のほうが重要であると考えられる. これらを考慮すると，各変数に重み付けを行うことにより，より正確な機能の抽象化が実現できると考えられる. 図 6 の $for3$ と $for4$ を例に考える. 実験において，コードブロック間の兄弟ブロック結合の閾値は 0.5 であるため 2 つの for 文は同一の機能に属すると判定されたが，閾値が 0.5 より大きい場合，異なる機能に属すると判定されてしまう．このとき $for3$ 内で使用されている変数に注目すると，変数 $codeRangeArray$ は代入が行われており，参照回数も変数 $codeRangeVector$ よりも多いため，変数 $codeRangeVector$ より重要度は高いことが分かる．このことを考慮し，変数 $codeRangeArray$ に高い重みを付けることにより，$for3$ と $for4$ の結合度は 0.5 よりも大きくなり，閾値が 0.5 より高い場合も同一の機能であると判定することができる．

本手法では 1 つのコードブロックは 1 つの機能に関連していることを前提に機能分割を試みている．しかし，実際には 1 つのコードブロックに複数の機能が実装されることがある．例えば，2 つの異なる機能に関連した変数の値を決定する際に，制御の流れが偶然同じであった場合，計算量を削減するため，1 つの制御文内で 2 つの異なる機能に関連した変数の値を決定することがある．保守性や可読性などを考慮すると，2 つの制御文に分割してメソッド抽出することが望ましいが，本手法は 1 つの制御文を複数の制御文に分割し，メソッド抽出対象とする機能は提供していない．このようなメソッド抽出リファクタリングには，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 [21] が有効である．

8　関連研究

統合開発環境 Eclipse は複数の自動リファクタリング機能を提供している [2]．自動化されるリファクタリングパターンには本手法が自動化するメソッド抽出リファクタリングも含まれる．しかし，Eclipse が自動化するのはコード変換のみであり，それ以外のプロセスはユーザが行わなければばならない．本手法はコード変換プロセス以外を自動化する．このため，本手法と Eclipse の自動化機能を併用することにより，本手法が提供するリファクタリングの全プロセスを自動化することができる．

丸山は基本ブロックラスシングを用いてメソッド抽出範囲を自動で特定する手法を提案した [21]．この手法はユーザにより指定された着目変数に依存関係を持つコード断片をスライシングにより既存メソッドから抜き出し，その断片を新規メソッドとしてまとめる．この手法はプログラムスライシングを応用しているため複数の処理が絡み合ったコード断片を別々のメソッドとして抽出することのできる有効な手法である．しかし，この手法は本手法と異なり，1 つの変数にしか着目していない．メソッドの機能と変数はつねに 1 対 1 に対応するわけではないため，1 つの変数を指定しただけでメソッドとして抽出すべき機能を指定できるわけではない．

2.4 節で述べたように，Ott らはスライスベースの凝集度メトリクスを定式化している．さらに，Krinke は Ott らの定義したメトリクスを応用し，文単位で計測可能なスライスベースの凝集度メトリクスを提案している [7]．また，適用事例により文単位の凝集度メトリクスが低い文はメソッド抽出の対象となると述べている．

Murphy-Hill らはリファクタリング自動化ツールが満たすべき要件を調査している [12]．具体的には，メソッド抽出リファクタリングを自動化する 3 つのツールを作成し，作成したツール同士の比較や既存のツールとの比較を行うために複数の被験者を対象とした実験を行った．その結果をもとに，Murphy-Hill らはリファクタリング支援ツール作成のためのガイドラインを提案した．本手法が対象とする，メソッド抽出範囲の特定に関しては次の 3 つの条件をのべている．

- 軽快なツール
- 例外的なコードフォーマットへの対応
- 特定のタスクに特化した能力

本手法は単一のメソッドのみを解析することにより抽出範囲を特定するため非常に軽快であり，ソースコードの選択も自動で行われるため，例外的なフォーマットをユーザが気にする必要はない．また，本手法はメソッド抽出リファクタリングに特化した支援手法である．以上のことより，本手法は Murphy-Hill らが提唱するメソッド抽出範囲の特定機能が満たすべ

き条件は全て満たしている．なお，メソッド抽出候補の識別に関しては，Murphy-Hill らのガイドラインでは述べられていない．

9　むすび

本論文ではリファクタリングに必要な労力を削減するため，メソッド抽出リファクタリングを支援する手法を提案した．具体的には次の 2 つのことを行った．

- メソッド内のコードブロックを機能要素，使用されている変数をデータ要素と見なし，メソッドの構成要素の協調度合を示すソフトウェアメトリクス COB を提案した．また適用事例から COB を用いることにより，LOC やサイクロマチック数などでは識別できないメソッド抽出の候補を識別できることを示した．
- COB を用いることによりメソッド抽出リファクタリングの対象として検出されたメソッドから，新規メソッドとして抽出すべき範囲を識別するための支援手法を提案した．また適用事例を用いて，提案手法を用いたメソッド抽出リファクタリングの例を示した．

本論文で提案したソフトウェアメトリクスや手法を用いることにより，メソッド抽出リファクタリングの対象となるメソッドの特定や新規メソッドとして抽出する範囲の決定に必要な労力を削減することができると考えられる．

今後の課題としては次の内容があげられる．

- 本研究で提案した手法はメソッド抽出範囲の識別を行うが，ソースコードを書き換え，メソッドとして抽出する作業は行わない．実際にメソッドを抽出する作業は統合開発環境 Eclipse にリファクタリング機能として実装されている．一方，Eclipse のリファクタリング機能はメソッド抽出範囲の識別は行わない．このため，本研究で提案したメソッド抽出範囲識別手法は Eclipse が提供するリファクタリング機能と併用することが可能である．しかし，5 章で紹介したシステムはスタンドアロンのツールとして実装されている．そこで本手法を Eclipse プラグインとして実装し，より実用的なツールとすることを考えている．Eclipse プラグインとして実装することにより，Roberts らが提唱するリファクタリングツールが満たすべき条件 [15](開発環境への統合，Undo 機能の実装など) も満たすことができる．また，より実装的なツールにするには Murphy-Hill らが提案するリファクタリング支援ツール作成のためのガイドラインも参考にする必要がある．
- 6 章で示した実験は単一のアプリケーションのみを対象としている．本研究で提案したメトリクスや手法の汎用性を示すにはさまざまな種類のアプリケーションを対象として実験を行い，より定量的に有効性を証明する必要がある．

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝致します.

本論文を作成するにあたり，常に適切な御指導，御助言を頂きました 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 松下 誠 准教授に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を頂きました 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 助教に心から感謝致します.

本論文を作成するにあたり，適切な御助言を頂きました 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 助教に心から感謝致します.

本研究を通して，様々な御協力を頂きました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 吉田 則裕 氏に深く感謝致します.

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

参考文献

[1] S. Chidamber and C. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 25(5):476–493, Jun 1994.

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

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

[4] B. Henderson-Sellors. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[5] 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, Australia, Mar 2008.

[6] S. Horwitz, J. Prins, and T. Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 146–157, San Diego, California, USA, Jan 1988.

[7] J. Krinke. Statement-Level Cohesion Metrics and their Visualization. In *Proc. of the 7th International Working Conference on Source Code Analysis and Manipulation*, pp. 37–48, Paris, France, Sep 2007.

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

[9] T. Mens and T. Tourwe. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb 2004.

[10] T. Meyers and D. Binkley. An Empirical Study of Slice-based Cohesion and Coupling Metrics. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–27, Dec 2007.

[11] T. Miyake, Y. Higo, and K. Inoue. A Metric-based Approach for Reconstructing Methods in Object-Oriented Systems. In *Proc. of the 6th International Workshop on Software Quality*, pp. 53–58, Leipzig, Germany, May 2008.

[12] 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, Leipzig, Germany, May 2008.

[13] P. Oman and S. L. Pleeger. Applying Software Metrics. *IEEE Computer Society Press*, 1997.

[14] L. M. Ott and J. J. Thuss. Slice-based metrics for estimating cohesion. In *Proc. of the 1th International Software Metrics Symposium*, pp. 71–81, Baltimore, MD, USA, May 1993.

[15] D. B. Roberts. Practical Analysis for Refactoring. Doctoral Thesis. UMI Order Number: AAI9944985, University of Illinois at Urbana-Champaign.

[16] D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–267, Jun 1997.

[17] C. Seguin and D. Li. JRefactory. `http://jrefactory.sourceforge.net/`.

[18] F. Simon, F. Steinbruchkner, and C. Lewerentz. Metrics Based Refactoring. In *Proc. of the 5th European Conference on Software Maintenance and Reengineering*, pp. 30–38, Lisbon, Portugal, Mar 2001.

[19] W. Stevens, G. Myers, and L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

[20] M. Weiser. Program slicing. In *Proc. of the 5th International Conference on Software Engineering*, pp. 439–449, San Diego, CA, USA, Mar 1981.

[21] 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.

[22] 三宅, 肥後, 井上. メトリクス計測プラグインプラットフォーム MASU の開発. ソフトウェアエンジニアリング最前線, pp. 63–70, Sep 2008.

[23] 仁井谷, 石尾, 井上. プログラムスライシングを用いた機能的関心事の抽出手法の提案と実装. 第 9 回プログラミングおよびプログラミング言語ワークショップ論文集, pp. 173–186, Mar 2007.