NAIST-IS-MT1151088

修士論文

スライスを用いた分割に基づくプログラム理解支援手法

平山 力地

2013年2月7日

奈良先端科学技術大学院大学
情報科学研究科 情報科学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士 (工学) 授与の要件として提出した修士論文である。

平山 力地

審査委員：

飯田 元 教授	（主指導教員）
関 浩之 教授	（副指導教員）
市川 昊平 准教授	（副指導教員）
吉田 則裕 助教	（副指導教員）

スライスを用いた分割に基づくプログラム理解支援手法*

平山 力地

内容梗概

近年のソフトウェア開発では，ソフトウェアの保守や拡張に多くの時間が費やされている．この保守に費やされる時間は，ソフトウェア開発全体にかかるコストの 3 分の 2 となっており，中でもソースコードを読み理解する作業は多くの時間を要する．そのため，ソースコードを読み理解する作業を支援し，時間を削減するための手法が求められている．

大規模ソフトウェアのソースコードを開発者が読む時，各ソースコードの先頭から 1 行ずつ読んでいき，行ごとの役割を理解した上で各メソッドの役割を理解し，そこからシステム全体をボトムアップに理解していくことは困難である．そのため開発者はソースコードを読む際，ソースコードの先頭から 1 行ずつ読んでいくのではなく，初めにドキュメントなどを読み全体的なシステムの理解から始め，その次に各メソッドの役割をトップダウンに理解していく．メソッド内のソースコードを理解する際も同様であり，まずコメントなどを読み，そのメソッドの役割を理解した上で，メソッド内のソースコードを単一の機能を実現するコード片に区切りながら，各コード片が実現している機能を理解し，メソッド全体を理解していく．しかし，多くのソフトウェア開発企業では，コメントやドキュメントの無いレガシーソフトウェアの存在が問題になっており，そのようなソフトウェアのソースコードはボトムアップに読まざるを得ず，より時間を要することになる．

本稿では，単一の機能を実現するコード片単位で開発者に提示することで，開発者のソースコードの理解を支援する手法を提案する．ソースコード内の単一の機能を実現するコード片を開発者に提示することで，コメントやドキュメントが無くて

* 奈良先端科学技術大学院大学 情報科学研究科 情報科学専攻 修士論文, NAIST-IS-MT1151088, 2013 年 2 月 7 日.

も，開発者はソースコードを読み始めるべき場所がわかる．ソースコードを機能ごとに分割・提示するために，凝集度という，モジュール内の各要素がどの程度協調して動作しているのかを表す度合いを利用する．本研究では，凝集度の高いコード片の集合を１つの機能と仮定し，ソースコードを機能ごとに分割・提示する．凝集度の高いコード片の集合は，プログラムスライスに基づく凝集度を用いて求める．またコード片を対象とした凝集度を求めるために，新たにコード片を対象とした凝集度メトリクスを定義した．

提案手法をツールとして実装し，開発者が手作業で求めた機能と対応付けたコード片と，ツールが出力したコード片を照合した．その結果，これらコード片は多くの場合に重複していることがわかった．

キーワード

プログラム理解，凝集度，プログラムスライシング，ソフトウェア保守

Slice Based Segmentation for Program Comprehension*

Rikichi Hirayama

Abstract

During software development, software maintenance is generally time consuming and therefore highly costly. Studies show that the time spent for software maintenance is two-thirds of the overall time spent for the software development, with long periods of time devoted to reading and understanding the source code. Therefore, there is a need for methods to ease this kind of tasks.

When the developers read the source code of large-scale software, it is difficult to read it line and line and understand the whole system in a bottom-up way. An alternative to this is using a top-down approach; when the developers read the source code, they first read documents to understand the whole system, and then, they understand the functions of each program method. In both approaches, the developers do the same for understanding a method. First, they read the comment. Next, they try to understand the functions of the method. Third, they understand the functions of code fragments in the method. Finally, they understand the whole method. However, old software that does not have comments and documentation become a problem for many software developers, and they have to spend a lot of time to read the source code using a bottom-up approach.

In this thesis, I propose an approach to aiding developers at this kind of software maintenance tasks. The proposed approach is based on dividing source

* Master's Thesis, Department of Information Science, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT1151088, February 7, 2013.

code into code fragments, each of which implements a single function. I assume that these fragments are easier to understand, even without comments and documentation. First, cohesion metrics were used to divide source code into fragments, presumably, implementing a single function. Then, slice based metrics were used to detect cohesive code fragments. Last, new metrics were defined to compute the cohesion of the code fragments.

I developed a tool that can detect automatically code fragments that implement a single function. I evaluated my tool using single function code fragments defined by developers. Results indicated that the tool could detect code fragments similar to ones defined by the developers.

Keywords:

Program comprehension, cohesion metric, program slicing, software maintenance

目次

図目次

表目次

1. はじめに

近年のソフトウェア開発では，ソフトウェアの保守や拡張に多くの時間が費やされている．この保守に費やされる時間は，ソフトウェア開発全体にかかる時間の 3 分の 2 となっており，中でもソースコードを読み理解する作業は多くの時間を要する [18, 3, 14]．そのため，ソースコードを読み理解する作業を支援し，時間を削減するための手法が求められている [6]．

大規模ソフトウェアのソースコードを開発者が読む際，各ソースコードの先頭から 1 行ずつ読んでいき，行ごとの役割を理解した上で各メソッドの役割を理解し，そこからシステム全体をボトムアップ理解していくことは困難である．そのため開発者はソースコードを読む際，ソースコードの先頭から 1 行ずつ読んでいくのではなく，初めにドキュメントなどを読み全体的なシステムの理解から始め，その次に各メソッドの役割などをトップダウンに理解していく．メソッド内のソースコードを理解する際も同様であり，まずコメントなどを読み，そのメソッドの役割を理解した上で，メソッド内のソースコードを単一の機能を実現するコード片に区切りながら，各コード片が実現している機能を理解し，メソッド全体を理解していく．以下のようなソースコードは，短時間で理解できると考えられる．

- ドキュメントやコメントが充分に書かれている
- モジュール化が充分に成されている

しかし古いソフトウェアには，ドキュメントやコメントが書かれておらず，モジュール化も不充分なソースコードが存在し，多くのソフトウェア開発企業で問題となっている．そのようなソースコードをトップダウンに理解することは困難である [14, 4]．理解が困難なソースコードには，単一のメソッド内に複数の機能を実現するコード片が存在することが多い [12]．

そこで本研究では，ソースコードを解析し単一の機能を実現するコード片単位で開発者に提示することで，開発者のソースコードの理解を支援する手法を提案する．またソースコードは機能ごとに排他的に分割できるものではなく，複数の機能に含まれるステートメント (ソースコード内の命令文) が存在すると考えられる．

そのためソースコードを単純に分割するのではなく，ステートメントのオーバーラップを考慮した機能候補の検出を行う．本研究では，凝集度の高いコード片の集合を 1 つの機能と仮定し，凝集度の高いコード片の集合を探すことでこれを実現する．ソースコード内の各要素間の依存関係からプログラム依存グラフを作成し，このプログラム依存グラフから凝集度の高いノードの集合を探し出す．凝集度を測るためには，プログラムスライスに基づいた凝集度メトリクスを利用する．しかし既存のプログラムスライスに基づく凝集度メトリクスは，メソッド全体を対象として定義されているため，コード片を対象とした凝集度を求めることができない．そこで本手法に適用するために，新たにコード片を対象とした凝集度メトリクスを定義する．そして，プログラムスライスに基づいた凝集度メトリクスを利用し，プログラム依存グラフから凝集度の高いコード片集合を探し出すことで，1 つの機能を実現しているコード片を見つける．

また凝集度の高いコード片の集合は，ソースコードが実現している最小の機能を表しており，開発者の考える 1 つの機能を実現しているコード片としては，小さすぎる傾向にあると考えられる．そこで，一般的に 1 つの機能と考えられているステートメントの集合を，依存関係に関係なくコード片として検出を行った．例えば，類似したステートメントが連続しているコード片や，変数の宣言・初期化文の組み合わせからなるコード片は，一般的に 1 つの機能と考えられるコード片である [15]．そして，凝集度メトリクスを利用して得られたコード片に組み込むことで，開発者の考える 1 つの機能を実現しているコード片に，より類似したコード片を出力できた．

これらの手法をツールとして実装し，実際にソースコードで実験を行った．複数の機能を含んでいるとされているメソッドに対して実験を行い，開発者が手作業で機能と対応付けたコード片と，ツールによって求めたコード片を比較した．その結果，これらコード片は多くの場合に重複していることがわかった．

2. 関連研究

本章では，本研究に関連する既存の研究を述べる．2.1 節でプログラムスライスについて述べ，2.2 節にてプログラムスライスに基づいた凝集度メトリクスについて述べる．2.3 節と 2.4 節では，既存のプログラム理解支援手法を提案している論文について述べ，2.5 節で既存研究の問題点を述べる．

2.1. プログラムスライス

プログラムスライスとは，各ステートメント間の依存関係に基づいて表された，ソースコード内の特定の処理に関わるステートメントの集合のことである [16]．ステートメントとは，ソースコード内の命令文のことであり，基本的に“;”で区切られた文が 1 つのステートメントとなる．図 1 の例では，4 つのステートメントが得られる．05〜07 行目は 1 つの処理であるため，行は違うが 1 つのステートメントとして扱われる．このように，ソースコードの 1 行が，そのまま 1 つのステートメントになるとは限らない．

この各ステートメント間の依存関係を用いて，グラフを作成したものがプログラム依存グラフである．図 3 は，図 2 のソースコードを簡単なプログラム依存グラフで表したものである．プログラム依存グラフは有向グラフであり，各ノードが各ステートメントを表しており，各エッジがステートメント間の依存関係を表している．エッジの向きは依存関係の向きを表しており，例えば図 3 では，05 行目のノードは 03 行目のノードへ依存している．

この各ステートメント間の依存関係には，データ依存とコントロール依存が存在する．データ依存とは 03 行目と 05 行目の間にあるような変数を介した依存関係を指す．05 行目の変数 s の計算結果は，03 行目の代入文にて変数 s に代入された値に依存している．またコントロール依存とは，04 行目と 05 行目の間にあるような制御に関連した依存関係を指す．05 行目が実行されるか否かは，04 行目の判定結果 (ここでは i ＜＝ a) に依存している．さらに 04 行目と 05 行目には，変数 i によるデータ依存が存在している．これは 05 行目の計算に用いられている変数 i が，04 行目で定義・代入されているためである．このように，2 つのステートメント間

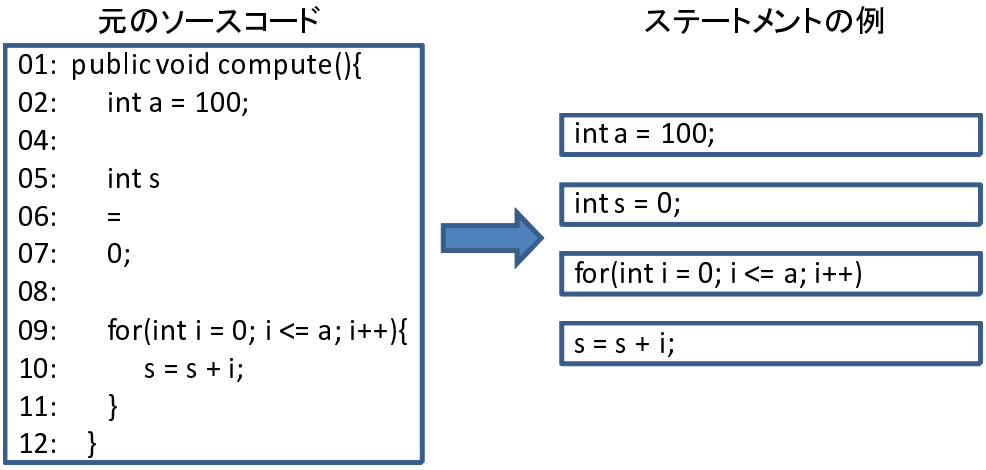


図 1　ステートメントの例

に複数の依存関係が存在することもある．

またプログラムスライスを得るためには，スライス基点を定義する必要がある．スライス基点は，一般にステートメントと変数で定義され，そのステートメントと変数は任意に決めることができる．ただし変数は，そのステートメントに含まれる必要がある．

プログラムスライスには，バックワードスライスとフォワードスライスの 2 種類に分類できる．同一のスライス基点から，この 2 種類のプログラムスライスを得ることができる．バックワードスライスとは，スライス基点として定義したステートメントと変数の計算・実行結果に影響を与えるステートメントの集合である（図 2)．フォワードスライスとは，スライス基点として定義したステートメントと変数の計算・実行結果が影響を与えるステートメントの集合である（図 4)．

図 2 はソースコードと，それに含まれるバックワードスライスの例である．青色で示されている配列 f に関するプログラムスライスは，10 行目と配列 f で定義されているスライス基点から得られたものである．このプログラムスライス (02，07，08，09，10 行目) は，10 行目の配列 f の計算結果に関わるステートメントの集合となっている．赤色で示されている変数 s に関するプログラムスライスのスライス基点は，05 行目と変数 s で定義されている．緑色で示されている変数 i に関するプロ

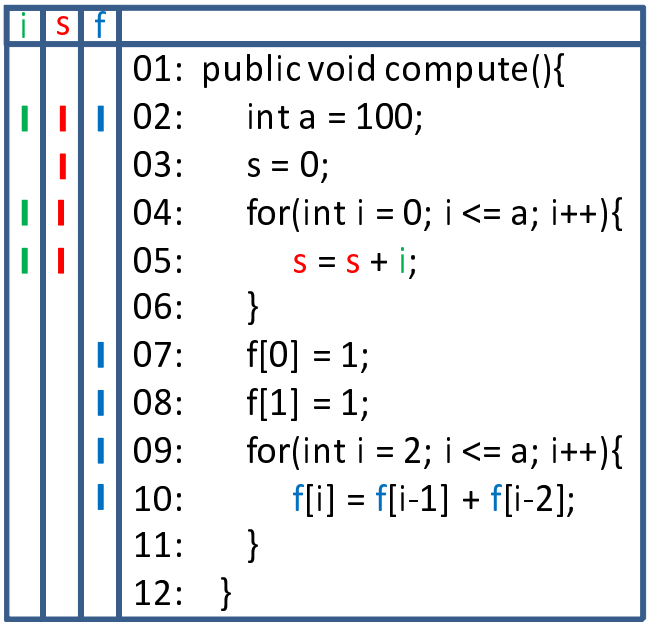


図 2　ソースコードとバックワードスライスの例

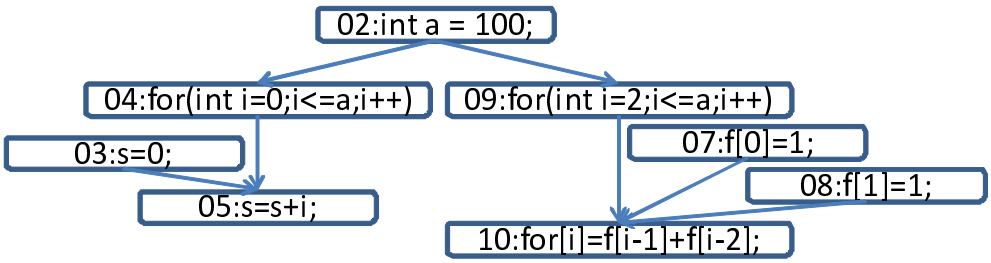


図 3　プログラム依存グラフの例

グラムスライスのスライス基点は，05 行目と変数 i で定義されている．この変数 s と変数 i に関するプログラムスライスの例のように，スライス基点として定義したステートメントが同一であっても，同時に定義する変数が異なると，得られるプログラムスライスが変わる．

図 4 はソースコードと，それに含まれるフォワードスライスの例である．青色で示されている変数 a に関するプログラムスライスは，02 行目と変数 a で定義され

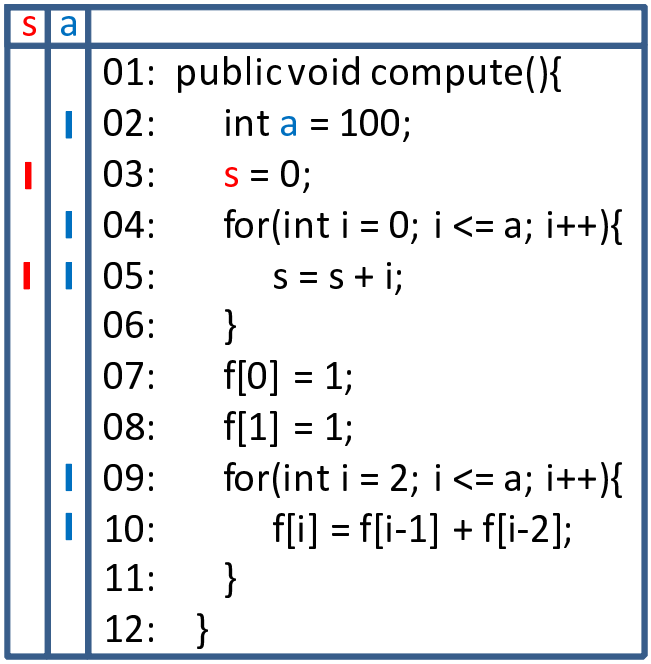


図 4　フォワードスライスの例

ているスライス基点から得られたものである．このプログラムスライス (02，04，05，09，10 行目) は，02 行目の変数 a の計算結果が影響を与えるステートメントの集合となっている．

これらのプログラムスライスは，プログラム依存グラフから容易に求めることができる．スライス基点となるステートメントが定まれば，それに対応するプログラム依存グラフ上のノードが定まる．そのノードから到達可能なノードの集合がフォワードスライスであり，そのノードに到達可能なノードの集合がバックワードスライスである [5]．またバックワードスライスは，特定の変数を計算するために必要なステートメントの集合等のソースコード内の特定の処理の流れを検出することができる．そのため，メソッド内に実装されている機能を検出することができると考えられる．よって本研究で用いるプログラムスライスは，説明が無い限りバックワードスライスとする．

2.2. スライスを用いた凝集度メトリクス

凝集度メトリクスはソースコードメトリクスの一種であり，ソースコードの品質を測るためのものである．本節で紹介する凝集度メトリクスはプログラムスライスを利用して，凝集度（対象メソッドの各文が, どの程度協調し合って実装されているかを表す尺度）を調べる．また本論文では，スライスを用いた凝集度メトリクスのことを，これ以降はスライスベースドメトリクスと記述する．一般に凝集度が高いメソッドの各ステートメントは互いに依存関係を持っており，1 つの機能を実現するためのメソッドであると考えられる [17].

スライスベースドメトリクスの代表的なものに Weiser の定義した凝集度メトリクスが存在する [16]．Weiser の定義した凝集度メトリクスは Tightness, Overlap, Coverage, Parallelism, Clustering の 5 種類があるが，中でも Overlap, Coverage，Tightness の有用性が高いとされている [10, 11]．ここでは本研究で用いる Overlap，Coverage，Tightness について説明し，各凝集度メトリクスの計算例を示す．またバックワードスライスをスライスベースドメトリクスの計算に用いる場合，対象メソッドの出力ステートメントをスライス基点とすることが一般的である [12]．この出力ステートメントというのは，戻り値を返すステートメント (Return ステートメント) だけでなく，フィールド変数などへの代入を行っているステートメントも含む．そのため計算例に用いるソースコード（図 5）では，05 行目の変数 s への代入を行っているステートメントと，10 行目の配列 f への代入を行っているステートメントを，このメソッドの出力ステートメントとし，計算を行った．なお 03，07，08 行目のステートメントもフィールド変数への代入を行っているステートメントだが，ここでは計算の簡略化のために省略している．

凝集度メトリクスの計算を行うメソッドを M，メソッドに含まれるプログラムスライスの総数を V_o，x 番目のプログラムスライスを SL_x，全てのスライスに共通するステートメントの集合を SL_{int} としたとき，Overlap の値は，

$$Overlap(M) = \frac{1}{V_o} \sum_{x \in V_o} \frac{|SL_{int}|}{|SL_x|} \tag{1}$$

で表される．これは，対象メソッドに含まれる全てのスライスに共通するステート

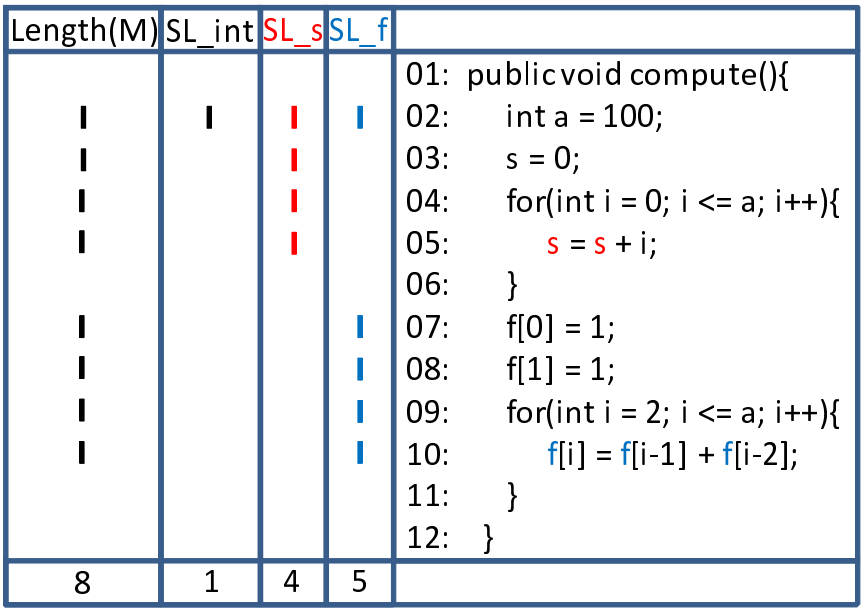


図 5　凝集度メトリクスの計算例

メントの数を，各スライスに含まれるステートメントの数で割ったものを足し合わせて，スライスの数で割ったものである．Overlap は各スライスに含まれる共通部分の割合を表している．Overlap が高い場合，そのメソッド内の各ステートメントは互いに高い依存関係を持っている可能性がある．

凝集度メトリクスの計算を行うメソッドを M，メソッドに含まれるプログラムスライスの総数を V_o，x 番目のプログラムスライスを SL_x，メソッドの長さを $length(M)$ としたとき，Coverage の値は，

$$Coverage(M) = \frac{1}{V_o} \sum_{x \in V_o} \frac{|SL_x|}{length(M)} \tag{2}$$

で表される．これは，対象メソッドに含まれる全てのスライスに含まれるステートメントの数の平均を，対象メソッド全体の長さで割ったものである．Coverage はメソッドに対して，意味のあるスライスの割合を表している．Coverage が低い場合，そのメソッドは複数の機能を含んでいる可能性がある．

凝集度メトリクスの計算を行うメソッドを M，全てのスライスに共通するステー

トメントの集合を SL_{int}，メソッドの長さを $length(M)$ としたとき，Tightness の値は，

$$Tightness(M) = \frac{|SL_{int}|}{length(M)} \tag{3}$$

で表される．これは，対象メソッドに含まれる全てのスライスに共通するステートメントの数を，対象メソッド全体の長さで割ったものである．Tightness が高い場合，そのメソッドの全てのスライスは同じ機能を実現するために実装されている可能性がある．

これらの凝集度メトリクスの計算結果は，図 5 の例では Overlap が 9/40，Coverage が 9/16，Tightness が 1/8 となっている．このように図 5 の例では，スライスベースドメトリクスの値は，どれも低くなることがわかる．これはメソッド内部が，変数 s を計算する箇所と変数 f を計算する箇所に，明らかに分かれているためであり，モジュール化が正しく成されていないためであると考えられる．このように凝集度メトリクスの値は，そのメソッドに含まれる機能の協調度合いを表している．

2.3. 使用変数に着目したソースコードの機能分類

木下は，各ステートメントの使用する変数に着目し，使用する変数が類似したコード片を 1 つの機能候補と見なし提示することで，ソースコードの理解支援を行った [9]．使用している変数に着目し，メソッド単位の凝集度を測るメトリクスは LCOM(Lack Of Cohesion in Methods) や COB(Cohesion Of Blocks) が存在している [2, 19]．使用する変数が類似したコード片を見つけるために，凝集度メトリクス COB を拡張し，任意のソースコードの領域に対して凝集度を求めることができるメトリクスである，COCP と NCOCP を定義した．ソースコード領域の機能別分類は，次の 3 ステップで行われる．

ステップ 1　ソースコードから構文木を作成
ステップ 2　構文木から，機能要素と呼ばれるコード片の抽出
ステップ 3　機能要素から，機能別に分類されたコード片の抽出

ステップ 1 では，ソースコードから構文木を作成するが，作成される構文木は木下が独自で定義した構文木となっている．ステップ 2 では，構文木を基に機能要素と呼ばれるコード片を抽出する．抽出されるコード片は，構文木上で近い位置にあり，かつ使用している変数が類似しているステートメントの集合となっている．ステップ 3 では，ステップ 2 で作成した各機能候補が使用している変数を調べ，使用している変数が類似している機能候補をまとめて，1 つの機能候補として扱う．この手法では同一の機能を実現しているコード片が別メソッドにある場合でも，1 つの機能候補として同時に検出可能である．

この手法を実装したツールを作成し，タンパク質の構造情報をもとにその可視化と三次元形状モデルへの変換を行うソフトウェア Chem3D3 のソースコードを対象に実験を行った．開発者が手動で機能を分析した結果と，ツールを用いて自動で分析した結果を比較した．その結果，開発者が考える機能と同様に，ソースコードを機能別に分類可能なことを確認した．

2.4. 空行の自動挿入による可読性の改善手法

Wang らは，ソースコードに適切に空行を入れることにより，ソースコードの可読性を改善した [15]．開発者が空行を入れる頻度の高い箇所，また Java のコーディング規約 ([13]) などで空行を入れた方が良いと定められている箇所を分析し，自動で空行を入れるアプリを開発した．適切な空行は可読性を上げるだけでなく，コメント挿入箇所の指標にもなるとされている．

この論文では，一般的に一つの機能であるとされるコードブロックを，いくつか定義している．Syntactically-Same Blocks は論文中で定義されているコードブロックの 1 つで，構文的に類似したステートメントが連続しているコードブロックを表している．図 6 は，Syntactically-Same Blocks の例である．変数が宣言・定義されているステートメントの集合であり，どれも構文的に類似したステートメントとなっている．

この手法を実装したツールを作成し，OSS のソースコードを対象に実験を行った．実験は，評価者がソースコードを読み手動で空行を入れた箇所と，ツールを用いて自動で空行を入れた箇所を比較した．その結果，一般的に空行を入れるべき箇所の 88.9% をツールでカバーできていた．

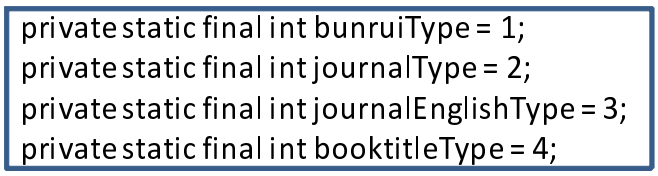


図 6 Syntactically-Same Blocks の例

2.5. 既存研究の問題点

2.3 節では，各ステートメントの使用する変数に着目したソースコードの機能分類手法について述べた．しかしこの手法は，ステートメント間の依存関係を考慮しておらず，処理の流れを考慮した機能の候補検出ができない．そのため，違う変数に値を渡した時などは，その処理を追うことができず，同一の機能に含まれていても違う機能として出力してしまう．また機能要素というコード片単位で，使用している変数を調べ機能候補を検出しているため，ステートメント単位での機能候補の検出ができない．

2.4 節では，ソースコードに適切に空行を入れることにより，ソースコードの可読性を改善する手法について述べた．この手法は，ソースコード上で隣り合ったステートメント間の依存関係のみを考慮している．依存関係の解析にプログラムスライスは使っておらず，ソースコード上で離れた箇所に存在するステートメント間の依存関係を得ることはできない．そのため，ソースコード上で離れた箇所に存在する複数のステートメントを提示することができない．この手法はソースコードの可読性を改善するための手法であり，機能候補を抽出することで開発者の理解を支援するという目的には適さないと考えられる．

3. 提案手法

本章では，プログラム依存グラフとプログラムスライス，凝集度メトリクスを利用した機能候補の検出手法について述べる．本研究では，ソースコード内の機能候補箇所の特定のために，スライスベースドメトリクスを使用する．2.2 節で述べたように，凝集度は，そのメソッドに含まれる機能の協調度合いを表している．そこでメソッド内の任意のコード片を対象に凝集度メトリクスを計測し，そのコード片の有する機能の数を調べる．そのコード片の凝集度が高ければ，そのコード片は単一の機能を実現するためのコード片である可能性が高いためである．逆に，凝集度が低いコード片は，複数の機能の実装を含んだコード片である可能性が高く，開発者のプログラム理解支援のために提示するコード片としては不向きである．

解析の手順は，大きく分けると次の 3 ステップに分かれている．

ステップ 1　解析を行うメソッドからプログラム依存グラフを作成

ステップ 2　ヒューリスティックを用いた機能抽出を行い，プログラム依存グラフに反映

ステップ 3　プログラム依存グラフから機能候補コード片を検出

ステップ 1 のソースコードの解析，プログラム依存グラフの作成には MASU を用いる [20]．また，スライスベースドメトリクスの計算方法は 3.2 にて解説を行う．ステップ 2 は 3.3 にて解説を行う．ステップ 3 は 2 種類の手法があり，3.1 にて，それぞれ解説を行う．

3.1. 凝集度メトリクスによる機能候補の取得

本節では，凝集度メトリクスを用いて，機能候補を検出する手法について説明する．機能候補はコード片として検出される．凝集度メトリクスを用いることで，ソースコード内の各ステートメント間の，処理の流れを考慮した機能抽出が可能となる．凝集度メトリクスにはスライスベースドメトリクスである，Overlap，Coverage，Tightness を用いた．

3.1.1. 閾値を用いた機能候補コード片の取得

対象メソッドのプログラム依存グラフ内の，全ての部分グラフ (ノードの組み合わせ) に対して凝集度を測り，閾値を超える部分グラフを探す．そして閾値を超えた部分グラフに対応するステートメント集合を，1 つの機能を実現しているコード片として提示する (図 7)．閾値を調整することで，検出されるコード片の大きさを変えることができる．3.1.2 節で提案する手法に比べると，検出されるコード片の大きさを，より細かく変えることが可能である．しかし，この手法はノード数を n 個とすると，計算量が $O(2^n)$ となる．スライスベースドメトリクスはノード数やノードの組み合わせによって，事前に値を予想することが困難であり，全ての組み合わせを計算する必要がある．これは n の値が大きくなると，現実的な計算時間では解くことができない [7]．そのため 3.1.2 節にて，コード片同士の合成を前提とした計算量の少ない手法を提案する．

3.1.2. 機能候補コード片の合成

対象メソッドのプログラム依存グラフ全体から，全てのプログラムスライスを取得する．各プログラムスライス内のステートメントの集合を，コード片として扱う．そして，各コード片をスライスベースドメトリクスを用いて，合成を行う (図 8)．コード片を合成するかは，実際にコード片同士の合成を行った際に，凝集度が閾値を超えるかで判定を行う．図 8 の例では，コード片 A と B，A と D，C と D，C と E を合成した場合，合成後のコード片が閾値以上となる例を表している．判定後，実際にコード片同士の合成を行うが，この時，コード片 A と B と D を 1 つのコード片に，コード片 C と D と E を 1 つのコード片になるように合成を行う．そして，合成により生成された新たなコード片を加え，この合成手順ができなくなるまで繰り返す．最後に，作成されたコード片同士の包含関係を調べ，各コード片の大きさが最大となるコード片の組み合わせを，機能候補として開発者に提示する．

3.2. コード片を対象とした凝集度メトリクス

2.2 節で説明した既存のスライスベースドメトリクスは，単一のメソッド全体の凝集度を測ることを目的としている．そのため，単一のメソッド内の任意のコード

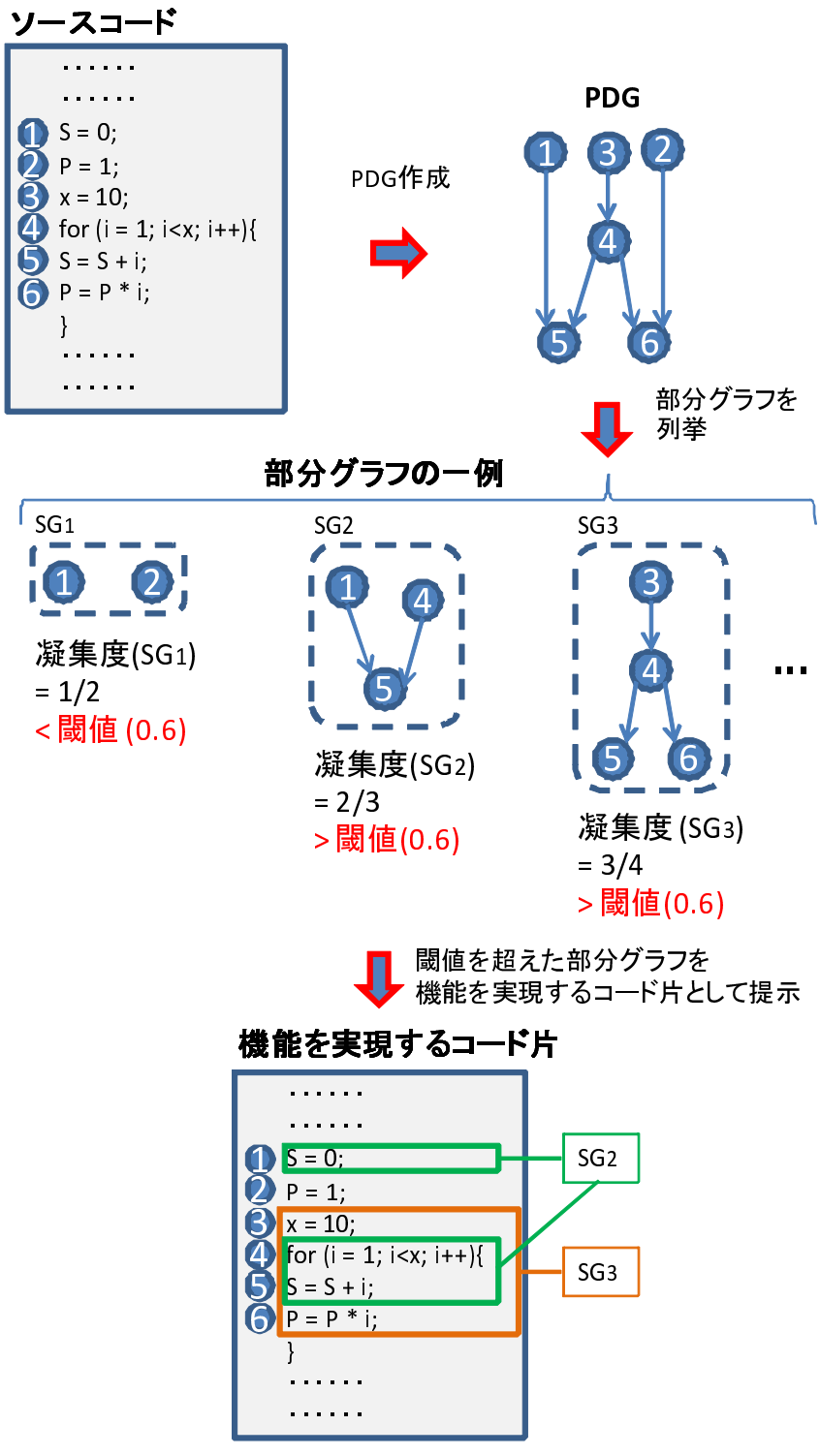


図 7　閾値を用いた機能候補コード片の取得

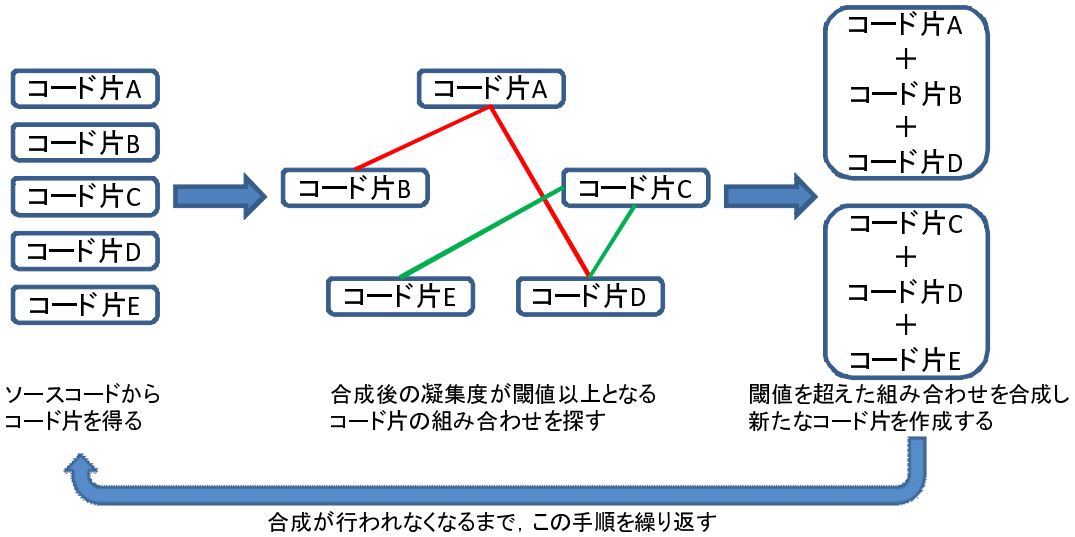


図 8　コード片の合成手順

片を対象とした凝集度を得ることはできない．そこでメソッドの一部の任意のコード片を対象とした，スライスベースドメトリクスを定義する．既存のスライスベースドメトリクスでは，計算に用いるプログラムスライスのスライス基点がメソッドの出力変数と定義されているため，任意のコード片を対象とした凝集度を得ることはできない．本研究では，単一メソッド内の任意のコード片のプログラムスライスを取得し，スライスベースドメトリクスの計算に利用する．

既存のプログラムスライスでは，メソッドの出力変数をスライス基点と定義しているが，これは，メソッド外（解析対象外）へ出力する変数をスライス基点としている，と言い換えられる．これを単一メソッド内の任意のコード片を対象とできるように拡張すると，プログラム依存グラフ上の以下のノードをスライス基点とすることに相当する．

(1) Return ステートメントや外部変数への出力を行う，解析対象範囲内のノード

(2) プログラム依存グラフ内の解析対象範囲外のノードへ依存関係を持つ，解析対象範囲内のノード

(3) プログラム依存グラフ全体で他のノードに依存関係を持たず，解析対象範囲に含まれるノード

(1) は一般的なスライス基点の定義である．一般的にバックワードスライスを取得する場合は，Return ステートメントや外部変数への出力を行っているステートメントを，スライス基点とする．

(2) は元のプログラムスライスの定義を元に，新たに定義を拡張したものである．解析対象範囲を 1 つのメソッドと考えた場合，解析対象範囲外のノードに依存関係を持つノードは，その解析対象範囲の出力と考えられる働きを持つからである．

(3) は本研究で新たに定義したスライス基点である．プログラム依存グラフの末端ノードは，元の一般的なバックワードスライスの定義では，スライス基点に選ばれることはない．しかしプログラム依存グラフの末端ノードには，出力文などソフトウェアの機能に関わるノードがあるため，本研究では出力文なども機能の一部として出力されるようにした．

また 1 つのスライス基点ノードから 1 つのプログラムスライスを出力するのではなく，複数のデータ被依存を持つスライス基点ノードは，各々のデータ被依存ごとに別のスライスを出力するようにした (図 9)．これは，一般的にスライス基点はステートメントと変数で定義されるが，プログラム依存グラフのノードのみをスライス基点とすると変数ごとのプログラムスライスを得ることができないためである．そのため図 9 のように，選ばれたスライス基点からデータ被依存ごとに複数のスライスを出力することで，変数ごとのプログラムスライスを得られるようにしている．

これらのスライス基点の定義を元に，解析対象としたコード片からプログラムスライスを得た時，得られたプログラムスライス同士に包含関係が存在することがある．図 10 は，メソッド全体を対象として，バックワードスライスを取得した例である．この時，03，05，07，08，10 行目は外部変数への出力を行っているステートメントであるため，スライス基点として選ばれる．また 05 行目は変数 s と変数 i に依存関係があるため，05 行目をスライス基点としたバックワードスライスは 2 個得られる．同様に 10 行目は変数 f と変数 i に依存関係があるため，10 行目をスライス基点としたバックワードスライスは 2 個得られる．このように，図 10 のソースコードから，本手法のスライス基点の定義ではプログラムスライスを 7 個得られるが，ほとんどのプログラムスライスが包含関係にあることがわかる．SL_fi，SL_f2，SL_f3 は SL_f1 に完全に包含されるプログラムスライスであり，SL_si，SL_s2 は

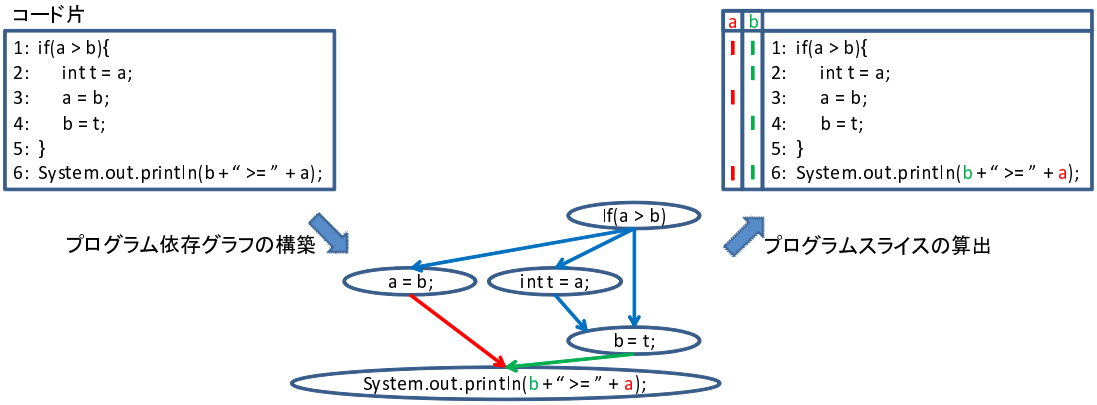


図 9　コード片を対象としたプログラムスライスの算出

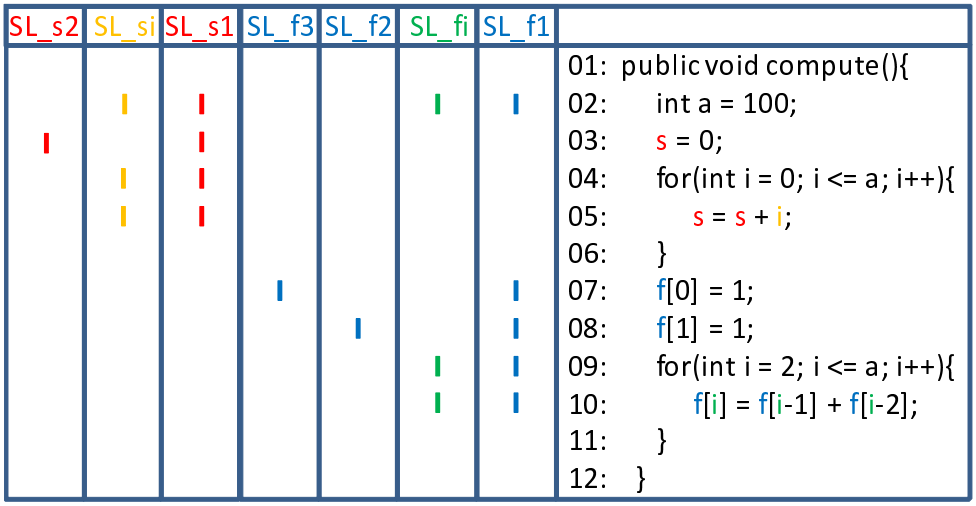


図 10　プログラムスライスの包含例

SL_s1 に完全に包含されるプログラムスライスである．他のプログラムスライスに完全に包含されるプログラムスライスは，限定的な処理の流れを表していたり，またプログラムスライス自体のサイズが非常に小さい傾向にある．小さいプログラムスライスを機能を実現するコード片の検出に用いても，開発者の想定するコード片より遥かに小さい物を検出してしまうことが多くなる．そのため本研究では，他のプログラムスライスに完全に包含されるプログラムスライスは，計算に利用しない．

またコード片の凝集度を測るために，2.2 節にて紹介した凝集度メトリクスの再定義を行った．Weizer の凝集度メトリクスでは，入力はメソッドと定義されていたが，本研究では入力にコード片を用いた凝集度メトリクスの再定義を行う．

この時，Overlap は，凝集度メトリクスの計算を行うコード片を CF，メソッドに含まれるプログラムスライスの総数を V_o，x 番目のプログラムスライスを SL_x，全てのスライスに共通するステートメントの集合を SL_{int} としたとき，

$$Overlap(CF) = \frac{1}{V_o} \sum_{x \in V_o} \frac{|SL_{int}|}{|SL_x|} \tag{4}$$

で再定義した．Overlap は，元の定義でも計算にメソッドの長さを用いないため，再定義後も式の右辺は変わらない．

Coverage は，凝集度メトリクスの計算を行うコード片を CF，メソッドに含まれるプログラムスライスの総数を V_o，x 番目のプログラムスライスを SL_x，コード片の長さを $length(CF)$ としたとき，

$$Coverage(CF) = \frac{1}{V_o} \sum_{x \in V_o} \frac{|SL_x|}{length(CF)} \tag{5}$$

と再定義した．

Tightness は，凝集度メトリクスの計算を行うメソッドを CF，全てのスライスに共通するステートメントの集合を SL_{int}，コード片の長さを $length(CF)$ としたとき，

$$Tightness(CF) = \frac{|SL_{int}|}{length(CF)} \tag{6}$$

と再定義した．

3.3. ヒューリスティックを用いた機能抽出

ステートメント間に依存関係が無くても，一般的に 1 つの機能を実現していると考えられるコード片が存在する [15]．本研究では，開発者の考える 1 つの機能を実現するためのコード片に，より類似したコード片を検出するため，ヒューリスティックを用いた機能の抽出を行う．

3.3.1. 変数宣言や初期化を含む文の抽出

変数の宣言文と，その初期化文の間には，基本的に依存関係は存在しない．しかし変数の宣言と初期化は，一般的に同時に行われる傾向にある．そのため，変数の宣言文と初期化文は，1 つのコード片として定義した．

3.3.2. 構文が類似した文の抽出

類似した構文のステートメントが連続していた場合，それは 1 つの機能を実現するためのコード片であると考えられる．図 11 は，1 つの機能を実現している類似したコード片の例である．この例では，088〜089 行目，092〜094 行目，097〜105 行目は，それぞれ 1 つの機能を実現していると考えられる．Switch 構文内の各 case の処理は，どれも類似した構文のステートメントの集合である．この各ステートメント間に依存関係は存在しないが，明らかに 1 つのまとまった処理であり，1 つの機能を実現していると言える．

また，ステートメント単位ではなく，コード片同士が類似した構文となっている場合がある．図 12 は，コード片同士が類似した構文となっている例である．062〜069 行目，072〜079 行目，082〜089 行目のそれぞれのコード片は，どれも構文的に類似したコード片となっている．構文的に類似したコード片が連続していた場合，本研究では，そのコード片の集合を 1 つの機能を実現しているコード片として定義した．図 12 の例では，062〜089 行目を 1 つの機能を実現しているコード片としている．

類似した構文のステートメントを判定するために，本研究では正規化を用いている．正規化とは，クラス名を A，メソッド名を B，という風に，ソースコード内の各ステートメントを単純な形に変形することで，類似判定を行う手法である [8]．図 13 は，正規化による構文解析の例である．3 つのステートメントは，正規化の結果，どれも“A.B(A.B);”と変形できるため，これらは類似したステートメントであると判断できる．

```
085: switch(intBunruiCode){
086:   case 1://学位論文（博士）
087:   case 2://学位論文(修士）j
088:       pBunken.setSchool(eForm.getSchool());
089:       pBunken.setSchoolEnglish(eForm.getSchoolEnglish());
090:       break;
091:   case 3://テクニカルレポート
092:       pBunken.setJournal(eForm.getJournal());
093:       pBunken.setJournalEnglish(eForm.getJournalEnglish());
094:       pBunken.setInstitution(eForm.getInstitution());
095:       break;
096:   case 4://論文誌
097:       pBunken.setEditor(eForm.getEditor());
098:       pBunken.setJournal(eForm.getJournal());
099:       pBunken.setJournalEnglish(eForm.getJournalEnglish());
100:       pBunken.setVolume(eForm.getVolume());
101:       pBunken.setNumber(eForm.getNumber());
102:       pBunken.setPublisher(eForm.getPublisher());
103:       pBunken.setPages(eForm.getPages());
104:       pBunken.setPublisherAddress(eForm.getPublisherAddress());
105:       pBunken.setImpactfactor(eForm.getImpactfactor());
106:       break;
```

図 11　類似したステートメントが連続している例

3.3.3.　if-else 文の抽出

if-else 文では，分岐先がそれぞれ別の機能であると考えることも可能だが，それぞれが全く違う機能であるとは考えにくい．if-else 文の分岐では，同一の条件式を元に分岐先が決まるため，条件式に応じた類似した処理をそれぞれの分岐に書いていると考えられるためである．本研究では，分岐先が類似した機能を実現していると仮定して，それを 1 つの機能を実現するコード片として定義した．

```
061: //保存先
062: LinkedHashMap<String,String> mapHozonsaki = new LinkedHashMap<String,String>();
063:
064: ArrayList<ParameterMasterSelect> hozonsakiList = controller.getMasterSelectCode(hozonsakiType,false);
065:
066: for(int iCnt=0; iCnt < hozonsakiList.size() ; iCnt++){
067:     mapHozonsaki.put(hozonsakiList.get(iCnt).getParamValue(), hozonsakiList.get(iCnt).getParamLabel());
068: }
069: inform.set("mapHozonsaki", mapHozonsaki);
070:
071: // 分類
072: LinkedHashMap<String,String> mapBunrui = new LinkedHashMap<String,String>();
073:
074: ArrayList<ParameterMasterSelect> bunruiList = controller.getMasterSelectCode(bunruiType,false);
075:
076: for(int iCnt=0; iCnt < bunruiList.size() ; iCnt++){
077:     mapBunrui.put(bunruiList.get(iCnt).getParamValue(), bunruiList.get(iCnt).getParamLabel());
078: }
079: inform.set("mapBunruiData", mapBunrui);
080:
081: // 分類削除対象
082: LinkedHashMap<String,String> mapBunruiDelete = new LinkedHashMap<String,String>();
083:
084: ArrayList<ParameterMasterSelect> bunruiDeleteList = controller.getMasterSelectCode(bunruiType,true);
085:
086: for(int iCnt=0; iCnt < bunruiDeleteList.size() ; iCnt++){
087:     mapBunruiDelete.put(bunruiDeleteList.get(iCnt).getParamValue(), bunruiDeleteList.get(iCnt).getParamLabel());
088: }
089: inform.set("mapBunruiDeleteData", mapBunruiDelete);
```

図 12　コード片同士が類似している例

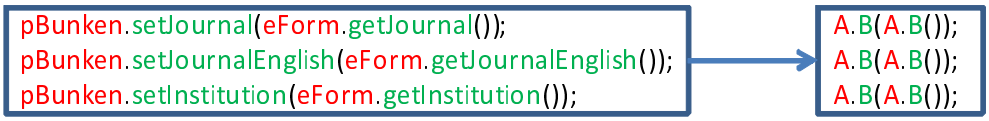


図 13　正規化

4. 実験

本章では，提案手法を元に実装したツールの概要と，その適用実験について述べる．

4.1. ツールの概要

ツールは Eclipse プラグインとして実装を行った．ソースコード内のメソッドを入力として与えると，そのメソッドに含まれる 1 つの機能を実現すると考えられるコード片を出力する．出力されるコード片の数はメソッドや，凝集度メトリクスの閾値によって変わる．

4.2. 実験設定

実験は，文献管理システムのソースコードを対象に行った．文献管理システムは，クラス数が 90 個，メソッド数が 1580 個，総 LOC が 20474 行のソースコードから成るプロジェクトである．この文献管理システムの，開発者が手作業で求めた機能と考えられるコード片 (図 14) と，ツールによって出力されたコード片の位置を比較する．位置の比較には GOOD 値，OK 値と呼ばれる指標を用いる [1]．この GOOD 値と OK 値は，コード片同士を比較するために作られた指標であり，正解集合に属するコード片に類似したコード片を，どれだけツールが出力できたか数値化することができる．本研究でも正解 (開発者が手作業で求めた機能と考えられるコード片) と，ツールの出力したコード片を比較する必要があるため，これを用いた．GOOD 値と OK 値を計算するには，まず $lineOverlap()$ と $lineContained()$ という関数を計算する必要がある．これは，2 つのコード片 C_1，C_2 を入力として定義した時,

$$lineOverlap(C_1, C_2) = \frac{|lines(C_1) \cap lines(C_2)|}{|lines(C_1) \cup lines(C_2)|} \tag{7}$$

$$lineContained(C_1, C_2) = \frac{|lines(C_1) \cap lines(C_2)|}{|lines(C_1)|} \tag{8}$$

で表される．$lineOverlap()$ は C_1 と C_2 の範囲が，どれだけ重なっているかを表しており，図 15 の例では 2/7 となる．$lineContained()$ は，C_1 の範囲を C_2 がどれだけ満たしているかを表しており，図 15 の例では，コード片 A を C_1，コード片 T を C_2 とした時，2/5 となる．また各メソッド内の，開発者が手作業で求めた機能と考えられるコード片を A_x，ツールによって出力されたコード片を T_y とする．x は 1 から n(開発者が分類したコード片の数) の値を取り，y は 1 から m(ツールの出力したコード片の数) の値を取る．この時，GOOD 値と OK 値は，

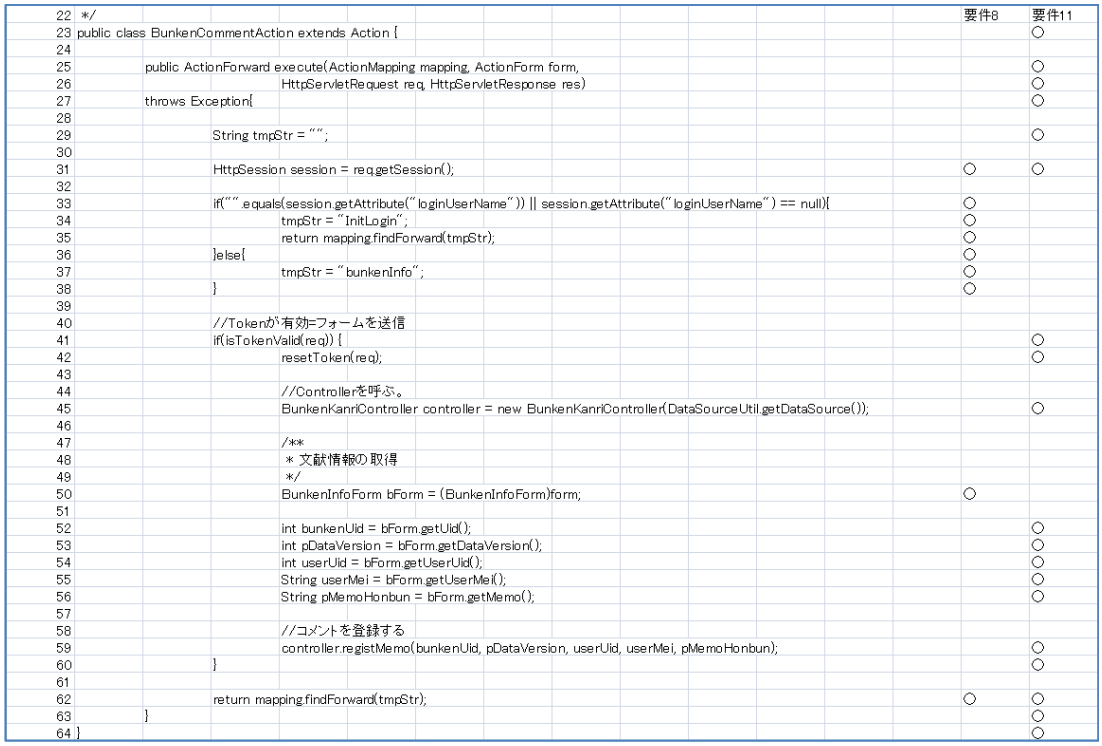
$$GOOD\ 値\ (A_x) = \max(lineOverlap(A_x, T_1), \ldots, lineOverlap(A_x, T_m)) \quad (9)$$

$$OK\ 値\ (A_x) = \max(lineContained(A_x, T_1), \ldots, lineContained(A_x, T_m)) \quad (10)$$

で表される．GOOD 値は，開発者が手作業で求めた機能と考えられるコード片に類似したコード片を，どれだけツールによって出力できているかを表している．OK 値は，開発者が手作業で求めた機能と考えられるコード片を，どれだけツールが 1 つの機能として出力できているかを表している．

また本研究は，複数の機能が実装されているメソッドの理解を支援するために提案されている．そこで開発者が，メソッド全体で 1 つの機能を実現していると判断したメソッドは，解析対象から外している．今回，実験対象としたソースコードでは，全体で 1 つだけの機能を実現しているメソッドは，5 行以下の小さいメソッドに多かった．これらを解析対象に含んでしまうと，正常な実験結果が得られなかったため，実験では除外している．

また開発者が，複数の機能が実装されていると判断したメソッドでも，他の機能に完全に包含されている機能は，解析対象に含めていない．本手法を実装したツールは，凝集度メトリクスの閾値によって，出力するコード片のサイズを変えることができる．しかし図 16 のような例では，開発者が定義したコード片集合と，同様のコード片集合をツールによって出力することはできない．これはツールが出力するコード片は，互いに包含関係ができないようにしているからである．包含関係を許容すると，一つのメソッドに対し多量のコード片が出力され，開発者が目を通すことができなくなると考えたためである．そのため，図 16 のような場合は，開発者の定義したコード片 2 とコード片 3 を除外している．開発者の定義したコード片 1 が，コード片 2 とコード片 3 を完全に包含するためである．また開発者の定義し



		要件8	要件11
22	*/		
23	public class BunkenCommentAction extends Action {		○
24			
25	public ActionForward execute(ActionMapping mapping, ActionForm form,		○
26	HttpServletRequest req, HttpServletResponse res)		○
27	throws Exception{		○
28			
29	String tmpStr = "";		○
30			
31	HttpSession session = req.getSession();	○	○
32			
33	if("".equals(session.getAttribute("loginUserName")) || session.getAttribute("loginUserName") == null){	○	
34	tmpStr = "InitLogin";	○	
35	return mapping.findForward(tmpStr);	○	
36	}else{	○	
37	tmpStr = "bunkenInfo";	○	
38	}	○	
39			
40	//Tokenが有効=フォームを送信		
41	if(isTokenValid(req)) {		○
42	resetToken(req);		○
43			
44	//Controllerを呼ぶ。		
45	BunkenKanriController controller = new BunkenKanriController(DataSourceUtil.getDataSource());		○
46			
47	/**		
48	＊文献情報の取得		
49	*/		
50	BunkenInfoForm bForm = (BunkenInfoForm)form;	○	
51			
52	int bunkenUid = bForm.getUid();		○
53	int pDataVersion = bForm.getDataVersion();		○
54	int userUid = bForm.getUserUid();		○
55	String userMei = bForm.getUserMei();		○
56	String pMemoHonbun = bForm.getMemo();		○
57			
58	//コメントを登録する		
59	controller.registMemo(bunkenUid, pDataVersion, userUid, userMei, pMemoHonbun);		○
60	}		○
61			
62	return mapping.findForward(tmpStr);	○	○
63	}		○
64	}		○

図 14　正解集合の例

た機能を除外した結果，メソッドに含まれる機能が 1 つだけになった場合，そのメソッドは解析対象に含まれなくなる．

解析に不適切なメソッド・機能を除外したところ，最終的に 29 個のメソッドが解析対象となった．この 29 個のメソッドは，各メソッドが複数の機能を含んでいると開発者に判断され，機能同士の包含関係がなく，充分な長さを持つメソッドである．

4.3.　実験結果と考察

Overlap，Coverage，Tightness の 3 つのメトリクスを用いて，凝集度メトリクスの閾値を変えて実験を行ったところ，GOOD 値と OK 値は表 1 のようになった．閾値が 0 の時は，メソッド全体が 1 つの機能を実現するコード片であると，機械的に提示することに相当する．そのため，3 つのメトリクスで実験を行ったが，

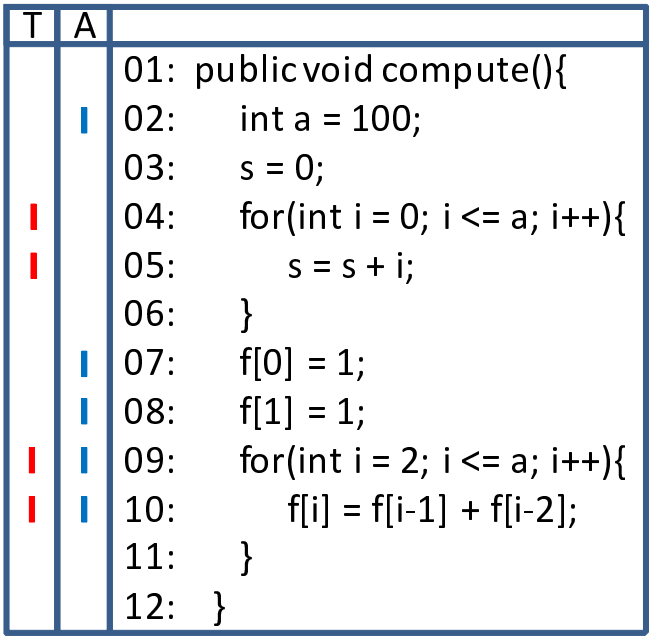


図 15　GOOD 値，OK 値の計算例

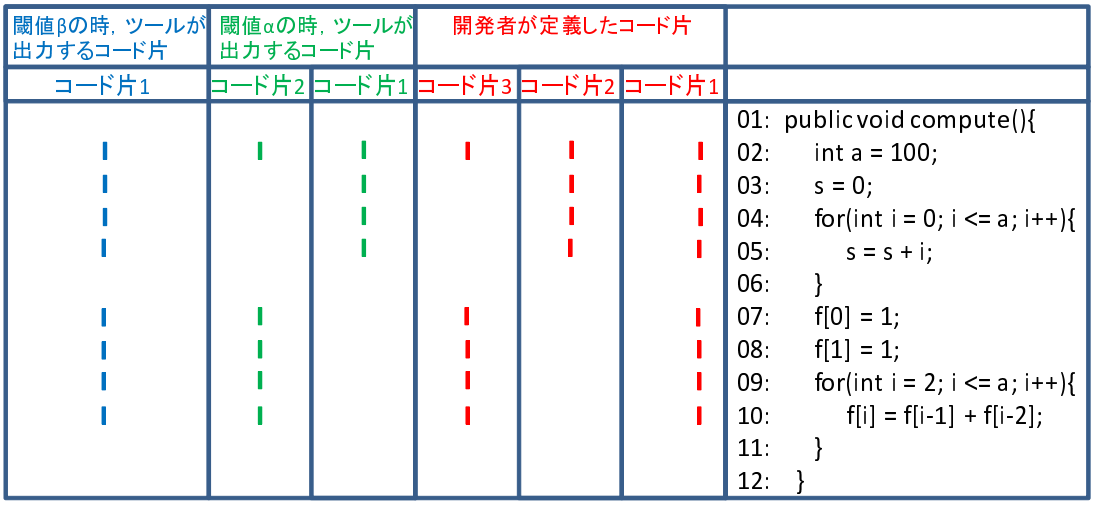


図 16　閾値によって変化するコード片の例

閾値が 0 の時に関しては，どれも同じ結果となっている．この時，OK 値は 1 となるが，GOOD 値が 0.526 と低い値を取っており，開発者の考えた機能に相当するコード片を提示しているとは言えない．また，閾値を 1 とし合成を行った後のコード片の，凝集度の分布を図 17, 図 18, 図 19 に示した．横軸 (c) は凝集度のデータ区間であり，最小値が 0，最大値が 1 となっており，0.01 単位でメモリが刻まれている．縦軸 (f) は，各凝集度の出現頻度を表している．例えば図 17 の例では，凝集度 (c) が 0.5 の時，頻度 (f) が 1000 を超えている．これは凝集度が 0.49〜0.5 となったコード片が，1000 個以上あったことを示している．

3 つのメトリクスを比較すると，Coverage を用いた際の GOOD 値が，他と比べ低くなる傾向が見られた．特に閾値が 0.1〜0.5 の時の GOOD 値は，閾値が 0 の時の GOOD 値と大きな差が見られなかった．これは，Coverage は 3.2 節で述べた式の定義上，コード片のサイズが変化しても凝集度が下がりにくいためである．そのため，合成の前後でコード片の凝集度に変化がなく，閾値を 0 にした時と同じように，全てのコード片が合成されてしまったからであると考えられる．

また閾値が 0.6 以上の時と，閾値が 0.5 以下の時を比較すると，Coverage を用いた際の GOOD 値が大きく変化している．これは Coverage の値の分布が，0.6 未満に集中しているためであると考えられる (図 18)．Tightness を用いた際は，GOOD 値は閾値に合わせて満遍なく変化しているが，閾値が 0.9 と 1 の時，結果に変化が無くなった．これは Tightness は，実験対象としたソースコードでは，0.9 以上の値を取ることが無かったためだと考えられる (図 19)．Overlap を用いた際は，GOOD 値は閾値に合わせて満遍なく変化している．Overlap は Coverage や Tightness に比べると，取り得る凝集度が 0.5 を中心に全体に分布しているため，閾値が結果に大きな影響を与えたからであると考えられる (図 17)．

凝集度メトリクスに Overlap を用いて，その閾値を 0.3 とした時，GOOD 値が最も高くなった．この時の GOOD 値は 0.7219，OK 値は 0.8682 となっており，開発者の定義した機能を実現するコード片の大部分を，ツールを用いて出力できている．図 20 は，ツールにて出力できた機能の例である．開発者が定義した機能を実現しているとされるコード片と，同じコード片を出力できている．また 2 つの違う機能を実現するコード片同士に，ステートメントのオーバーラップがあっても，それぞれのコード片で同一のステートメントを出力することができた．

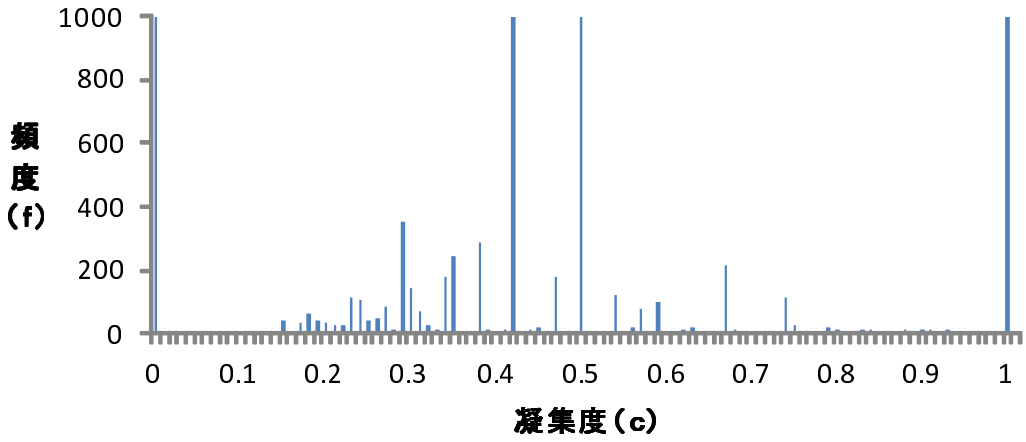


図 17　Overlap の分布

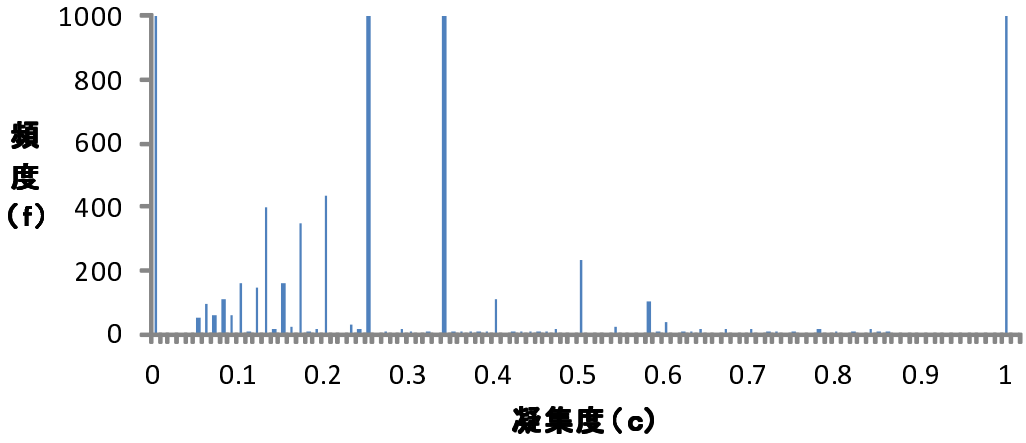


図 18　Coverage の分布

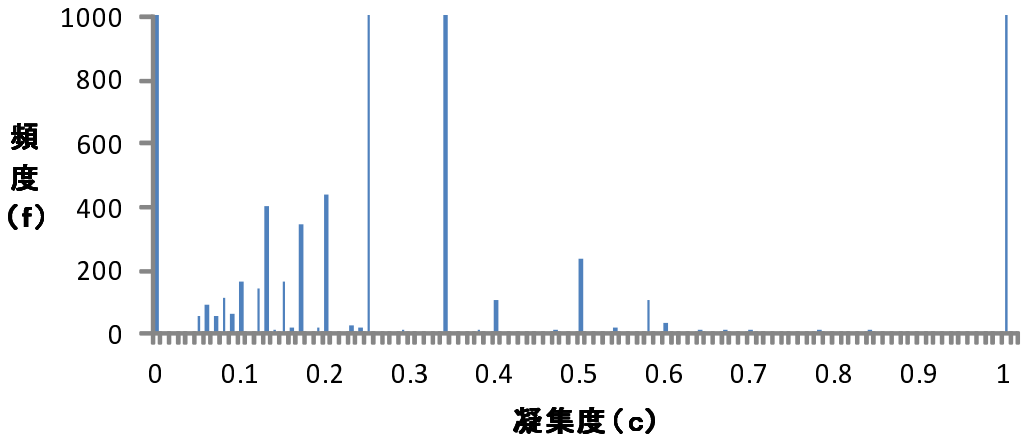


図 19　Tightness の分布

実験結果から，実験対象としたソースコードでは，使用する凝集度メトリクスは Overlap が最も適していた．Coverage と Tightness は，閾値を変化させても，得られるコード片の変化が少ない．そのため，凝集度メトリクスの閾値を調整することで，得られるコード片のサイズを調整する目的がある場合，用いる凝集度メトリクスは Overlap が妥当と言える．また各凝集度メトリクスが取る値の分布は，式の定義に依存しているため，対象のソースコードが変わっても値の分布に大きな変化はない．よって実験対象としたソースコード以外でも，メソッドに含まれる機能を抽出するという目的には，Overlap が最も適していると考えられる．

また Overlap を用いた時，閾値が 0.3 の時に最も良い結果となった．これは実験対象としたソースコードの，開発者が手作業で求めた機能と考えられるコード片に，大きいサイズのコード片が多く存在したためだと考えられる．メソッドの半分以上のサイズのコード片を機能と定義しているケースが多く，3 つ以上の機能を含んでいるとしていたメソッドは，ほとんど存在しなかった．また得られるコード片のサイズは，対象とするメソッドの規模や凝集性に依存する．しかし，得られるコード片の数は，対象とするメソッドの凝集性にのみ依存すると考えられる．凝集性の低いソースコードを対象とした場合，得られるコード片は増え，凝集性の高いソースコードを対象とした場合，得られるコード片は減る．そのため，実験対象と

	Overlap		Coverage		Tightness	
閾値	GOOD 値	OK 値	GOOD 値	OK 値	GOOD 値	OK 値
0	0.5263	1	0.5263	1	0.5263	1
0.1	0.7066	0.9405	0.5263	1	0.7070	0.8330
0.2	0.6943	0.9269	0.5289	0.9952	0.7071	0.7365
0.3	0.7219	0.8682	0.5274	0.9889	0.7075	0.7158
0.4	0.6833	0.7965	0.5294	0.9839	0.6999	0.7082
0.5	0.7030	0.7954	0.5235	0.9640	0.6897	0.6974
0.6	0.7071	0.7995	0.6842	0.8051	0.6878	0.6955
0.7	0.6948	0.7855	0.6999	0.7082	0.6743	0.6820
0.8	0.6860	0.7754	0.6860	0.6936	0.6720	0.6797
0.9	0.6747	0.7634	0.6720	0.6797	0.6712	0.6788
1	0.6735	0.7622	0.6712	0.6788	0.6712	0.6788

表 1　メトリクスごとの GOOD 値と OK 値

したソースコード以外でも，実験対象と凝集性が類似したソースコードであれば，閾値を 0.3 とすることで，類似した数のコード片を機能として得られると考えられる．また実験対象としたソースコードは，メソッドを跨いだ処理が少なく，ほとんどのメソッドが返り値を持っており，処理がメソッド内部で完結していると言えるメソッドが多かった．そのため，引数を受け取り，返り値を返すようなメソッドは，実験対象としたソースコード以外でも，今回の実験と同様に，有意な結果が得られると考えられる．しかし，メソッドを跨いだ処理が多様されるソースコードを対象とした場合は，今回の実験と同様の結果は得られないと考えている．

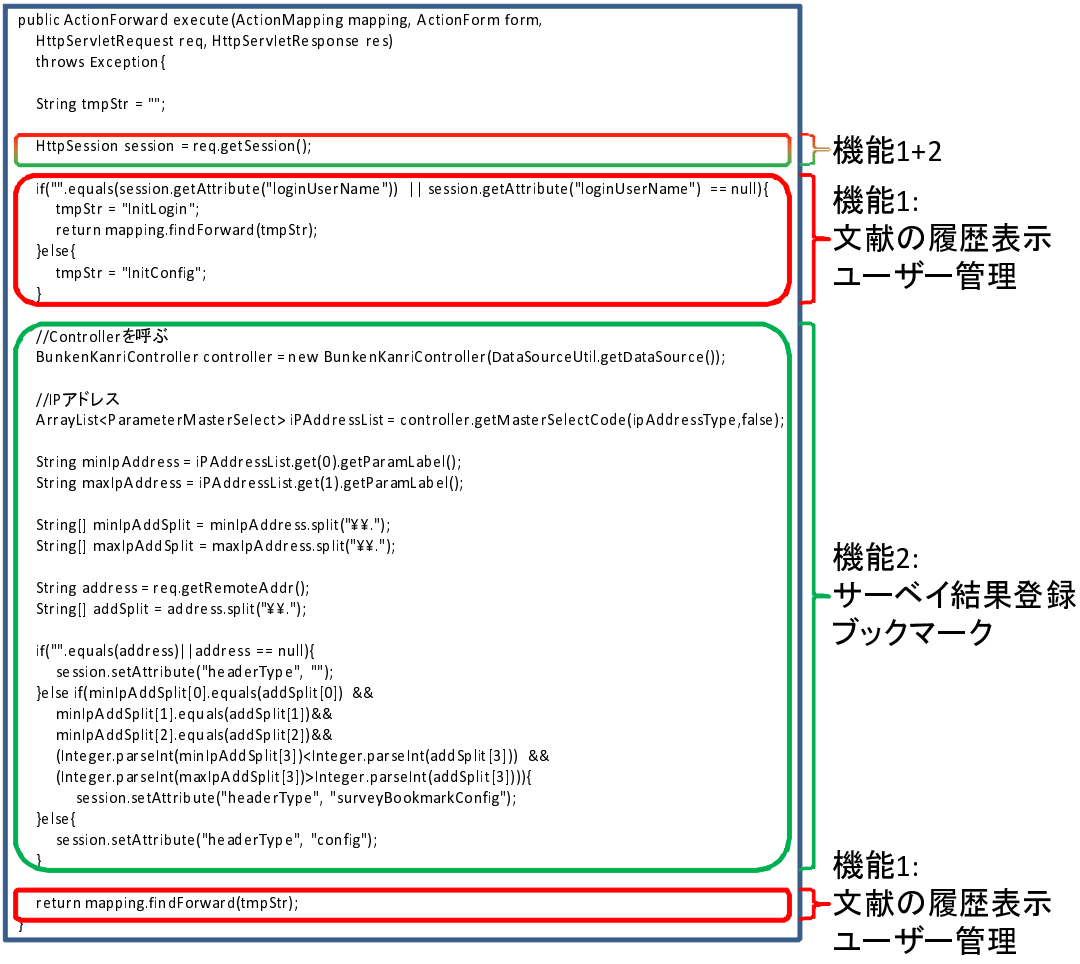

public ActionForward execute(ActionMapping mapping, ActionForm form,
HttpServletRequest req, HttpServletResponse res)
throws Exception{
String tmpStr = "";
HttpSession session = req.getSession();
if("".equals(session.getAttribute("loginUserName")) || session.getAttribute("loginUserName") == null){
tmpStr = "InitLogin";
return mapping.findForward(tmpStr);
}else{
tmpStr = "InitConfig";
}
//Controllerを呼ぶ
BunkenKanriController controller = new BunkenKanriController(DataSourceUtil.getDataSource());
//IPアドレス
ArrayList<ParameterMasterSelect> iPAddressList = controller.getMasterSelectCode(ipAddressType,false);
String minIpAddress = iPAddressList.get(0).getParamLabel();
String maxIpAddress = iPAddressList.get(1).getParamLabel();
String[] minIpAddSplit = minIpAddress.split("¥¥.");
String[] maxIpAddSplit = maxIpAddress.split("¥¥.");
String address = req.getRemoteAddr();
String[] addSplit = address.split("¥¥.");
if("".equals(address)||address == null){
session.setAttribute("headerType", "");
}else if(minIpAddSplit[0].equals(addSplit[0]) &&
minIpAddSplit[1].equals(addSplit[1])&&
minIpAddSplit[2].equals(addSplit[2])&&
(Integer.parseInt(minIpAddSplit[3])<Integer.parseInt(addSplit[3])) &&
(Integer.parseInt(maxIpAddSplit[3])>Integer.parseInt(addSplit[3]))){
session.setAttribute("headerType", "surveyBookmarkConfig");
}else{
session.setAttribute("headerType", "config");
}
return mapping.findForward(tmpStr);
}
機能1+2
機能1:
文献の履歴表示
ユーザー管理
機能2:
サーベイ結果登録
ブックマーク
機能1:
文献の履歴表示
ユーザー管理

図 20　機能分類結果

5. おわりに

本研究では，開発者のソースコードの理解を支援するために，ソースコードを機能ごとに分割・提示する手法を提案した．ソースコードの分割にスライスベースドメトリクスを用いることで，処理の流れを考慮した分割が可能になった．用いるスライスベースドメトリクスとして既存のメトリクスではなく，任意のコード片の凝集度を測れるメトリクスを新たに提案した．また，一般的に 1 つの機能と考えられている箇所を検出し，スライスベースドメトリクスと共に本手法に用いた．これによって，開発者の考える 1 つの機能を実現しているコード片に，より近いコード片を分割・提示できたと考えられる．実験では，実際に開発者が手作業で求めた機能と考えられるコード片と，ツールによって求められたコード片を比較し，本手法の有意性を確かめた．

今後は，実験対象とするソースコードが違っても，用いる凝集度メトリクスとして Overlap が適しているか確認する必要がある．また同様の凝集性を持つソースコードに対して，閾値を 0.3 とした時，対象のメソッドを，今回の実験対象と同じような数に分割できるか確認する必要がある．そのため規模や凝集性の違う複数のソースコードに対して，実験を行う必要があると考えられる．また本研究の手法では，複数メソッドに処理が分散している場合，それらを 1 つの機能として抽出することができない．より開発者の理解を支援するためには，メソッドを跨いだ機能の抽出を実装する必要があると考える．

謝辞

本研究を進めるに当たり，多くの方々に，ご指導，ご協力を頂きました．

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研究室 飯田 元 教授には，本研究の全過程において，多くの指導を賜りました．研究方針だけではなく，研究に対する姿勢，論文執筆，発表方法についても多くの御助言を頂きました．心より厚く御礼を申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 計算メカニズム学研究室 関 浩之 教授には，様々な場面で本研究に対し，貴重なご指導，ご助言を賜りました．心より感謝申し上げます．

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研究室 市川 昊平 准教授には，様々な場面で本研究に対し，貴重なご指導，ご助言を賜りました．心より感謝申し上げます．

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研究室 田中 康 特任准教授には，様々な場面で本研究に対し，貴重なご指導，ご助言を賜りました．心より感謝申し上げます．

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研究室 吉田 則裕 助教には，本研究を進めるに当たり，広範囲かつ多大な御助力を頂きました．研究方針だけではなく，研究に対する姿勢，論文執筆，発表方法についても多くの御助言を頂きました．心より感謝申し上げます．

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研究室 CAMARGO CRUZ Ana Erika 助教には，様々な場面で本研究に対し，貴重なご指導，ご助言を賜りました．心より感謝申し上げます．

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研究室 高井 利憲 特任助教には，様々な場面で本研究に対し，貴重なご指導，ご助言を賜りました．心より感謝申し上げます．

NTT データの伏田 享平氏には，本学在籍時に，多大なご助力をいただきました．また，研究への姿勢，について多くのアドバイスを頂戴いたしました．心より感謝申し上げます．

東洋大学 角田 雅照 助教， 福岡工業大学 戸田 航史 助教には本学在籍時に，研究のみならず，私生活においても多大な助言を頂きました．心より感謝申し上げます．

株式会社デンソーの木下 正喬氏，大阪大学 大学院情報科学研究科 後藤 祥 氏には，本研究について多くのアドバイスを頂戴いたしました．心より感謝申し上げます．

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研究室，ならびにソフトウェア工学研究室の皆様には，日頃より多大な御協力と御助言を頂き，公私ともに支えていただきました．ありがとうございました．最後に，日頃より私を励まし，温かく見守ってくれた家族に心より深く感謝します．

参考文献

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, Vol. 33, No. 9, pp. 577–591, 2007.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, Vol. 20, No. 6, pp. 476–493, 1994.

[3] Jr. Deimel and E. Lionel. The uses of program reading. *SIGCSE Bull.*, Vol. 17, No. 2, pp. 5–14, 1985.

[4] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, Vol. 29, No. 3, pp. 210–224, 2003.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, Vol. 9, No. 3, pp. 319–349, 1987.

[6] N. Gold and K. Bennett. Hypothesis-based concept assignment in software maintenance. *Software, IEEE Proceedings*, Vol. 149, No. 4, pp. 103–110, 2002.

[7] T. Jiang, N. Gold, M. Harman, and Z. Li. Locating dependence structures using search-based slicing. *Inf. Softw. Technol.*, Vol. 50, No. 12, pp. 1189–1209, 2008.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7, pp. 654–670, 2002.

[9] 木下正喬. プログラム理解のための凝集度に基づくコード片の分類手法. 奈良先端科学技術大学院大学 修士論文, NAIST-IS-MT0951038, 2011.

[10] J. Krinke. Statement-level cohesion metrics and their visualization. In *Proc. of SCAM*, pp. 37–48, 2007.

[11] M. O. Linda and M. B. James. Program slices as an abstraction for cohesion measurement. *Inf. Softw. Technol.*, Vol. 40, No. 11–12, pp. 691 – 699,

1998.

[12] T. M. Meyers and D. Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Trans. Softw. Eng. Methodol.*, Vol. 17, No. 1, pp. 2:1–2:27, 2007.

[13] Sun Microsystems. Code Conventions for the Java Programming Language, 1999. `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`.

[14] D. R. Raymond. Reading source code. In *Proc. of CASCON*, pp. 3–16, 1991.

[15] X. Wang, L. Pollock, and K. Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *Proc. of WCRE*, pp. 35–44, 2011.

[16] M. Weiser. Program slicing. In *Proc. of ICSE*, pp. 439–449, 1981.

[17] Edward Y., Larry L. C., 原田実 (訳), 久保未沙 (訳). ソフトウェアの構造化設計法. 日本コンピュータ協会, 1986.

[18] 独立行政法人情報処理推進機構 ソフトウェア・エンジニアリング・センター. ソフトウェア開発データ白書 2009. 日経 BP 社, 2009.

[19] 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. 92, No. 7, pp. 1071–1073, 2009.

[20] 三宅達也, 肥後芳樹, 楠本真二, 井上克郎. 多言語対応メトリックス計測プラグイン開発基盤 MASU の開発. 電子情報通信学会論文誌 D, Vol. 92, No. 9, pp. 1518–1531, 2009.