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

信学技報
TECHNICAL REPORT OF IEICE.

プログラム構造の簡略化によるメトリクス計測方法の改善

佐々木　唯† 　石原　知也† 　堀田　圭佑† 　畑　　秀明† 　肥後　芳樹†
井垣　　宏† 　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{s-yui,t-ishihr,k-hotta,h-hata,higo,igaki,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソフトウェア保守の分野において，サイクロマチック数は代表的な複雑度メトリクスとして良く用いられている．しかし，サイクロマチック数はソースコード中の分岐の数を表しているだけで内容は考慮していないため，サイクロマチック数が大きいからといって人が複雑だとみなすとは限らない．例えば，ソースコード中に if 文が繰り返し記述されている場合サイクロマチック数は増大するが，これらが単純な記述の繰り返しであれば，保守に影響を及ぼす複雑なソースコードであるとは考えにくい．本稿では，人が複雑だとみなすソースコードを識別するために，ソースコード中の繰り返し部分を簡略化してメトリクスを計測する手法を提案する．提案手法の有用性を確認するために，オープンソース・ソフトウェアに対してメトリクスを計測し，手法適用の有無による比較を行った．その結果，提案手法を適用して計測したメトリクスは，人間の主観による複雑度の評価に近いことが確認できた．
キーワード　実証的研究，複雑度メトリクス，メトリクス計測

Improving Software Metrics Measurement by Simplifying Program Structures

Yui SASAKI†, Tomoya ISHIHARA†, Keisuke HOTTA†, Hideaki HATA†, Yoshiki HIGO†, Hiroshi IGAKI†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
E-mail: †{s-yui,t-ishihr,k-hotta,h-hata,higo,igaki,kusumoto}@ist.osaka-u.ac.jp

Abstract　Cyclomatic complexity is used as a representative complexity metric in software maintenance. However, it does not always represent cognitive complexity. It is because cyclomatic complexity does not consider the contents of the branch but only count the number of paths in the source code. For example, repeated if-statements produce a high cyclomatic complexity value. Such structures are repetitions of simple instructions, which are not complicated and do not have a negative impact on software maintenance. In this paper, we propose a method to identify cognitive complexity by simplifying such repeated structures. We conducted a case study with open source software systems to reveal the usefulness of the proposed method, and then we confirmed that the metrics values with the proposed method had stronger correlation with human consideration.
Key words　Empirical Study, Complexity Metrics, Metrics Measurement

1. ま え が き

ソフトウェア工学におけるエンピリカルアプローチのための基礎的な技術として，メトリクスを用いたソフトウェアの計測手法が知られている．例えば，近年ソフトウェアリポジトリマイニングが盛んに行われ，フォールトプローンを含むモジュールの特定に有用な履歴メトリクスが多く提案されている．最近の研究では，フォールトプローン検出には伝統的なプロダクトメトリクスよりも履歴メトリクスが有用であることが示されている [1], [2]．プロダクトメトリクスとしてはソースコードの行数や複雑度などが存在し，これらを用いてリファクタリングすべきモジュールの特定などが行われている [3], [4]．

複雑度を表すメトリクスには様々な種類のものが存在する．McCabe のサイクロマチック数 [5] は，モジュールの制御パス数を表す複雑度メトリクスである．サイクロマチック数は伝統的に用いられるメトリクスではあるが，必ずしも複雑さを表す

とは限らないことが既存の研究で述べられている．オブジェクト指向プログラムにおいては，サイクロマチック数はクラス間の複雑度を考慮していないことが述べられており，Chidamber と Kemerer は，クラスの重み，結合度，凝集度などを含む 6 つの複雑度メトリクスを定義している [6]．また，Buse らはサイクロマチック数とソースコード可読性の相関が低いことを示している [7]．そのため，サイクロマチック数は必ずしも人が感じる複雑さを表現していないとも考えられる．更に，Jbara らは Linux カーネルに含まれるサイクロマチック数の値が高い関数を調査し，switch 文中に含まれる case 文や連続して記述された単純な if 文が多く含まれる関数は，サイクロマチック数が高くてもソフトウェアの品質に影響を及ぼすものではないと報告している [8]．また，pmccabe [9] など，サイクロマチック数の計測方法を拡張したツールはいくつか提案されているが，その有用性に関する実験は十分に行われていない．

これまでの研究において，サイクロマチック数が必ずしも人が感じる複雑さを表さない要因には，連続した case 文や if 文のように，ソースコード中に繰り返し記述された構造があるという共通点がある．これは行数の計測でも同じことがいえる．例えば GUI 部品の初期化を行う長いモジュールには代入文が繰り返し記述されており，行数が示すほどの規模の大きなモジュールであるとは限らない．既存のメトリクスを用いてソースコードを直接計測すると，上記のような繰り返し構造を持つモジュールがメトリクス値の高いモジュールとして検出される．そのため，ソフトウェア保守の妨げとなるような可読性の低い複雑な構造を持つモジュールの発見が難しくなる可能性がある．また，類似した記述による繰り返し構造は 1 つの機能的なまとまりを持っていると考えられ，行数やサイクロマチック数の値が高くても保守に影響を及ぼすものではないと考えられる．

そこで本研究では，ソースコード中の繰り返し構造を簡略化してメトリクスを計測する手法を提案する．提案手法の有用性を確認するために，約 13,000 のオープンソース・ソフトウェアに対して提案手法を用いたメトリクス計測を行った．その結果，従来のメトリクス値と比べてメトリクス値が下がるメソッドが多く存在することが分かった．更に，提案手法によって計測されるメトリクス値は人が感じる複雑さを表現していることを確認した．

以降，2 章ではこの研究の動機となる 2 つのメソッドを紹介する．3 章では，提案手法の中で用いる繰り返し構造の折りたたみについて説明する．4 章では実験内容と結果について述べ，5 章でその考察を行う．最後に 6 章を本稿のまとめとする．

2. 研 究 動 機

図 1(a) はあるソフトウェアに含まれるメソッドである．このメソッドには多くの if 文が含まれ，ネストの深い構造になっている．サイクロマチック数は 33 と高い値を持ち，複雑なメソッドであることが分かる．一方，このソフトウェアにはサイクロマチック数 112 のメソッドが存在する．図 1(a) のメソッドと比べてサイクロマチック数が約 3 倍であることから，このメソッドは非常に複雑な構造を持つと予想される．しかし，こ

```
  1: public Object getValoreIndirizzobenefattotrans(...) {
       ...
       if() { ... if() { ... if() { ... if() { ...
124:      if (soggetto != null) {
              ...
128:          else
129:         {
130:             ArrayIterator iter = ...;
131:             if (iter!=null && iter.size()>0) {
132:                 iter.reset();
133:                 IfIndirizzo recapito = ...;
134:                 if(...)
135:                     return ...;
136:                 else return null;
137:             }
138:             else
139:                 return null;
140:         }
141:       }
142:       else return null;
         } } } }
189: }
```

(a) ネストの深い構造を持つメソッド

```
  1: public int getColumnIndex(final String sColumnName)
  2: {
  3:     if (sColumnName.compareToIgnoreCase(...) == 0)
  4:         return NDX_TI_ID_TITOLO;
  5:     else if (sColumnName.compareToIgnoreCase(...) == 0)
  6:         return NDX_TI_ID_COMPAGNIA;
       ...
204:     else if (sColumnName.compareToIgnoreCase(...) == 0)
205:         return NDX_FI_DESC_FILIALE;
206:     return -1;
207: }
```

(b) 繰り返し構造を含むメソッド

図 1　サイクロマチック数の高いメソッドの例

Fig. 1　Examples of High Cyclomatic Complexities Methods

のメソッドの構造は図 1(b) に示すよう，単純な if-else 文が繰り返し記述されているのみで，複雑な構造であるとは考えにくい．これらの例が示す通り，サイクロマチック数は必ずしも複雑さを表すとは限らない．

本稿では，図 1(b) のようにソースコード中に繰り返し構造を含む場合，サイクロマチック数が従来の計測値よりも低く計測されるよう，繰り返し構造を折りたたむ前処理を行った上でメトリクスを計測する手法を提案する．提案手法では，抽象構文木 (Abstract Syntax Tree, **AST**) を用いてソースコード中の繰り返し構造を発見し，折りたたんで単一の構造として表現する．全ての繰り返し構造を折りたたんだソースコード（以下，簡略化されたソースコードと呼ぶ）に対して，サイクロマチック数などのメトリクス値が計測される．図 1 にある 2 つのソースコードに対して提案手法を適用すると，図 1(b) のサイクロマチック数は図 1(a) のサイクロマチック数よりも低く計測できる．

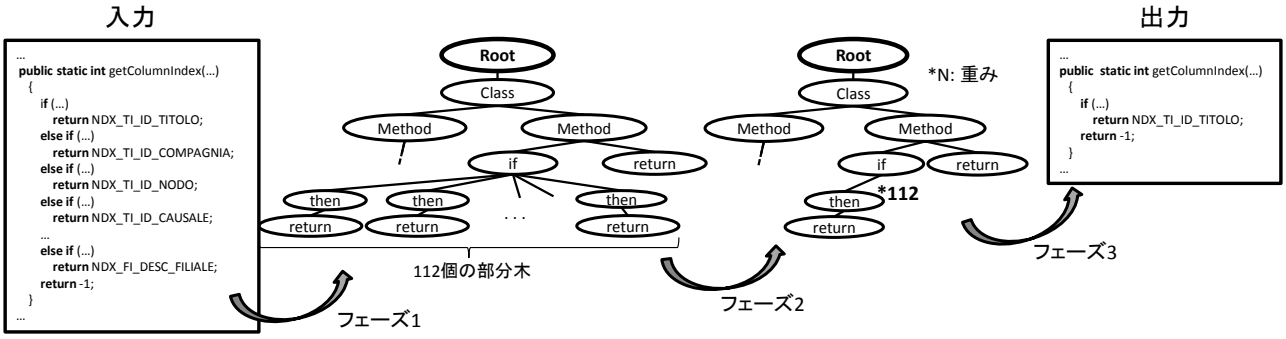


図 2　図 1(b) のソースコードに対する提案手法の流れ
Fig. 2　Proposed Method for Fig.1(b)

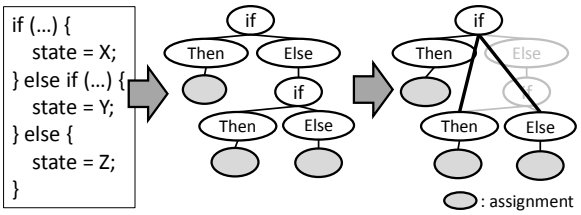


図 3　else-if 節の変形
Fig. 3　Transformation of else-if Statements

3. 提 案 手 法

本章では，簡略化されたソースコードを生成する手法を紹介する．本手法は以下の流れで実行される．

フェーズ 1.　入力されたソースコードから AST を構築する
フェーズ 2.　AST 中の繰り返し構造を折りたたたむ
フェーズ 3.　折りたたまれた AST を元に，簡略化されたソースコードを出力する

図 1(b) のメソッドを含むソースコードを例とした手法の流れを図 2 に示す．続いて，手法の詳細について説明する．

3.1　フェーズ 1．AST の構築

まず与えられたソースコードを読み込んで，AST を構築する．ただし，図 1(b) のように if 文の後に連なる else-if 節については，AST の変形処理を同時に行う．変形処理とは，AST 上で else 節を表すノードの直下に if 文を表すノードのみが存在する場合，これらのノードを削除し，削除された部分木の親ノードと子ノードを接続することを指す．この処理の流れを図 3 に示す．変形処理を行うことで，後に述べるフェーズ 2 において，連続する else-if 節を繰り返し構造であるとみなすことができる．

また，本手法ではメソッド中の文構造にのみ着目し，変数名や条件式といった文の詳細は考慮しない．そのため，本稿では AST にそれらの情報を表示していない．

3.2　フェーズ 2．繰り返し構造の折りたたみ

3.2.1　繰り返し構造の定義

AST 上の兄弟ノードは，ソースコード上の出現順による順序性を持っている．そこで，連続する兄弟ノードが類似していれば，それらの兄弟ノードからなる部分木の集合を繰り返し構造とみなす．ノードが類似しているかどうかは表 1 の基準で判断する．表 1 中の”重み”については，次節で述べる．

ここで，ソースコード中には複数種類の文による記述の繰り返しが存在する．例えば以下のソースコードは，代入文とメソッド呼び出し文という 2 種類の文による記述の繰り返しである．

```
comparator = new ObjectIdentifierComparator();
cb.schemaObjectProduced( this, "2.5.13.0", comparator );
comparator = new DnComparator();
cb.schemaObjectProduced( this, "2.5.13.1", comparator );
```

このように複数の文単位での繰り返しも AST 上で繰り返し構造と判断できるよう，複数種類の兄弟ノード列が AST 上で連続している場合も，繰り返し構造とみなす．

3.2.2　折りたたみ

繰り返し構造の折りたたみとは，繰り返し単位となる部分木（列）を 1 つだけを残して削除し，繰り返し回数を表す重みを根ノードに対して持たせることである．

繰り返し構造の中には入れ子になったものも存在する．例えば以下のソースコードは，メソッド呼び出し文の繰り返しと，それをネスト内に含む if 文の繰り返しが存在する．

```
if (null != storepass) {
    cmd.createArg().setValue(“-storepass”);
    cmd.createArg().setValue(storepass);
}
if (null != storetype) {
    cmd.createArg().setValue(“-storetype”);
    cmd.createArg().setValue(storetype);
}
```

このような場合，メソッド呼び出し文を先に折りたためるよう，AST の葉に近いノードほど先に折りたたみを行う．

3.2.3　フェーズ 2 の手順

以上を踏まえて，フェーズ 2 は次の手順で実行される．

表 1　ノードの類似基準
Table 1　Criteria for Similarity of Node

比較されるノード対	類似とみなす判定基準
共に子を持たない	文の種類と重みが同じ
共に子を持つ	ノードを根とする部分木が同形 かつ 対応する全てのノード対が類似

まず，AST を後順走査して兄弟ノードを取得する．続いて，兄弟ノード中の繰り返し構造を特定する．このとき，繰り返された文の単位が小さいものから順に判断し，兄弟ノード中に繰り返し構造が見つからなくなるまで再帰的に折りたたみを行う．

図 2 の場合，then 節を表すノードを根とする全ての部分木が繰り返し構造とみなされ，メソッド中の大部分の構造が折りたたまれた．

3.3 フェーズ 3. ソースコード生成

最後に，折りたたまれた AST から簡略化されたソースコードを生成する．このソースコードに対してメトリクスを計測すると，図 1(b) のメソッドはサイクロマチック数 2, 行数 6 という結果が得られた．一方，図 1(a) のメソッドは繰り返し構造をほとんど含まないため，メトリクス値はあまり変化しなかった．

4. 実　　験

提案手法を実装し，評価実験を行った．今回の実装は Java で記述されたソースコードのみを対象とした．本実験の対象は，UCI source code data sets [10] に含まれる約 13,000 のソフトウェアである．このデータセットの規模を表 2 に表す．これらのソフトウェアに含まれる全てのメソッドに対して，提案手法を適用した場合としなかった場合の，行数とサイクロマチック数を計測した．以降，これらのメトリクスを本稿では表 3 のように呼ぶ．また，ここで計測した行数は，ソースコード中の空行やコメント行を含まない．

この実験では，次の 2 つの項目について調査し，提案手法の有用性について確認する．

（1） 計測されるメトリクス値にどのような違いがあるか

（2） 提案手法によって計測されるメトリクスは，人が感じる複雑さを表しているか

次節以降では，上記の項目に対する実験内容と結果について述べる．

4.1 実験 1. 計測されるメトリクス値の比較

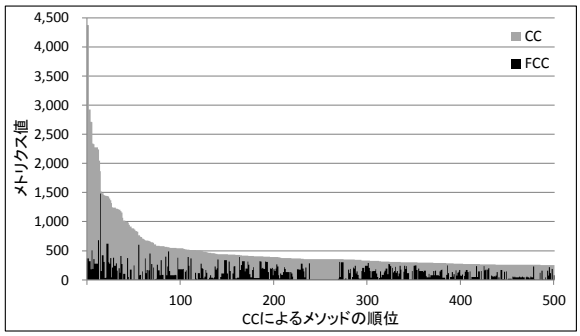
実験 1 では，提案手法適用の有無によるメトリクス値の違いをメソッドごとに調査する．全メソッド中，CC, LOC それぞれの値が大きいメソッド上位 500 個について，メトリクス値の違いを表したグラフを図 4 に示す．2 つのグラフともに，CC, LOC の値に比べて FCC, FLOC の値が大きく減少しているメソッドが多く見られ，特に従来の計測値の大きかったメソッドはその変化が顕著である．従って，提案手法を適用することで，計測されるメトリクスの値は大きく影響を受けることが分かる．

表 2　UCI source code data sets の概要
Table 2　Overview: UCI source code data sets

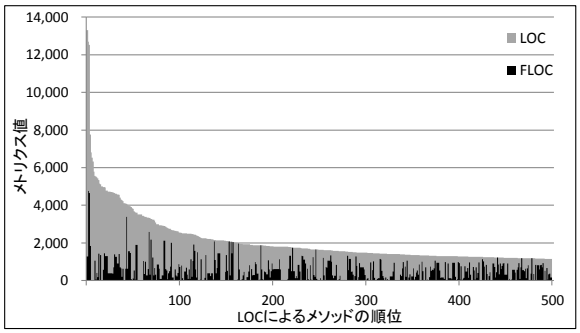
ソフトウェア数	13,193
メソッド数	18,366,094
総行数	361,663,992

表 3　計測対象メトリクス
Table 3　Target Metrics

	行数	サイクロマチック数
提案手法を未適用	LOC	CC
提案手法を適用	FLOC	FCC

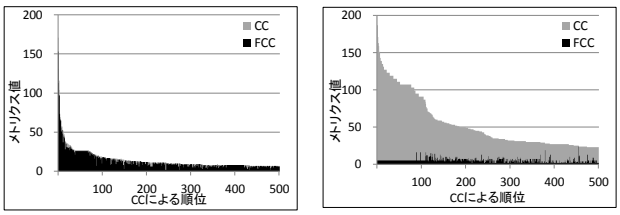


(a) サイクロマチック数



(b) 行数

図 4　メトリクス値の違い
Fig. 4　Difference of Metrics Values



(a) 違いの少ない例　　(b) 違いの多い例

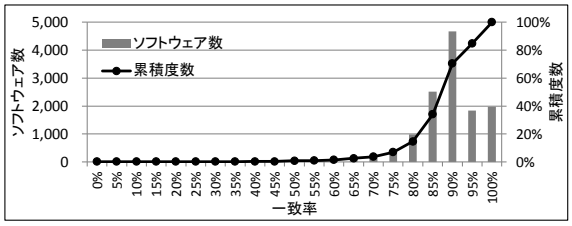
図 5　各ソフトウェアにおけるサイクロマチック数の違い
Fig. 5　Difference of Metrics Values on Each Software

また，このようなメトリクス値の違いをソフトウェアごとに調査したところ，図 5 に示すように，計測値にほとんど違いがないソフトウェアや，計測値が大きく異なるソフトウェアなど，ソフトウェアによって傾向は異なっていた．そこで，上記のような傾向をソフトウェアごとに得るため，各メトリクス値による上位 n 個のメソッド中に同じメソッドが含まれる割合 ”一致率” を定義し，ソフトウェアごとに計測した．サイクロマチック数の場合，一致率は以下のように定義される．

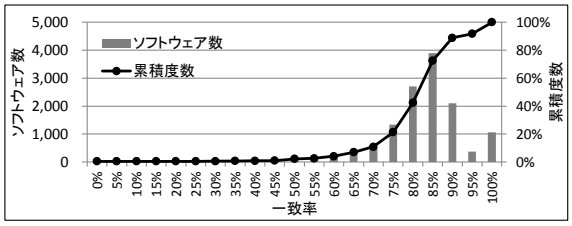
$$\text{一致率}\ (n) = \frac{|CC(n) \cap FCC(n)|}{n}$$

n は上位とみなすメソッドの数を表し，$CC(n)$ は CC の値による順位付け上位 n 個の集合を，$FCC(n)$ は FCC の値による順位付け上位 n 個の集合を表す．

全ソフトウェアに対する，サイクロマチック数，行数による一致率の分布を図 6 に示す．このグラフは一致率を 5%刻みで



(a) サイクロマチック数



(b) 行数

図 6　一　致　率
Fig. 6　Concordance Rate

分布したヒストグラムである．例えば一致率 80%の項目は，一致率が 80%以上 85%未満であるソフトウェアの数を表している．また，対象ソフトウェアの規模が様々であるため，ここでは上位とみなすメソッド数 n をソフトウェアに含まれる全メソッド数の 20%にあたる数と定めている．

このグラフから，どちらのメトリクスについても一致率の高いソフトウェアが多いことが分かる．しかし一致率が 100%であるソフトウェアは，サイクロマチック数の場合はソフトウェア全体の約 15%，行数の場合は約 8%であり，提案手法によって多くのソフトウェアでメトリクス値に上位のメソッドが変化するという結果を得られた．

4.2　実験 2. 被験者による複雑さ評価とメトリクス値の比較

実験 2 では，被験者による複雑さの判定を元に，提案手法が複雑なメソッドの特定に有用かどうか調査を行う．本実験では，本学コンピュータサイエンス専攻の教員・大学院生 8 名を被験者とし，次の手順で実験を行った．

（1）　対象ソフトウェア中の全メソッドについて，複雑か複雑でないかの評価を被験者が各自行う

（2）　ある被験者によって複雑だと評価されたメソッドを**正解メソッド**とし，そのメソッドの集合を**正解集合**とする（正解集合は被験者の数だけ得られる）

（3）　各メトリクス値によるメソッドの順位付け上位 20%中，正解メソッドがいくつ含まれるか比較する

被験者に対してメトリクス値等の情報は提示されておらず，ソースコードの閲覧によってのみ複雑かどうかが評価される．また，メソッド 1 つ 1 つを判断する被験者の負担が大きくならないよう考慮し，メソッド数 389 という規模がそれほど大きくないソフトウェア JCap を実験対象として選択した．このソフトウェアには，提案手法によって繰り返し構造とみなされ，計測されたメトリクス値が異なるメソッドが多く含まれている．各メトリクス値による順位付け上位 20%における一致率は，サイクロマチック数の場合は約 69%，行数の場合は約 67%である．

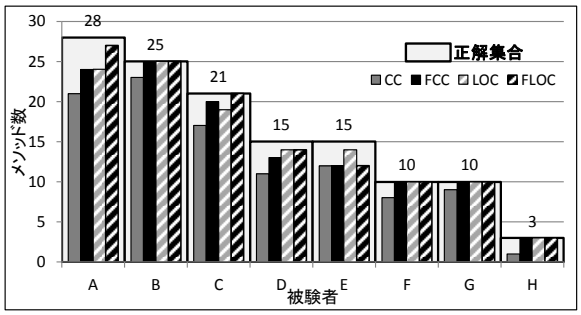
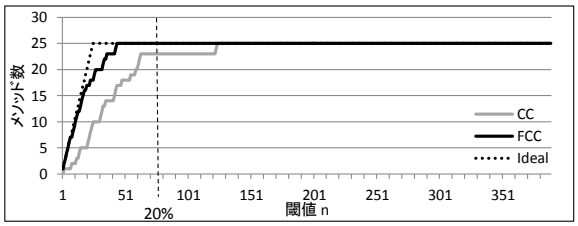
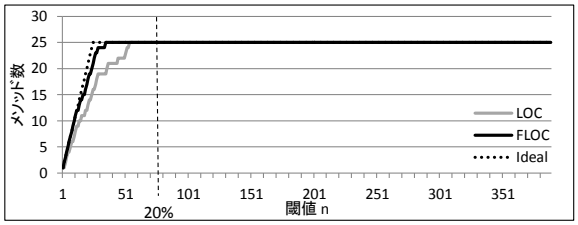


図 7　上位 20 パーセント中の複雑なメソッド検出数
Fig. 7　Comparison between Human Consideration and Metrics



(a) サイクロマチック数

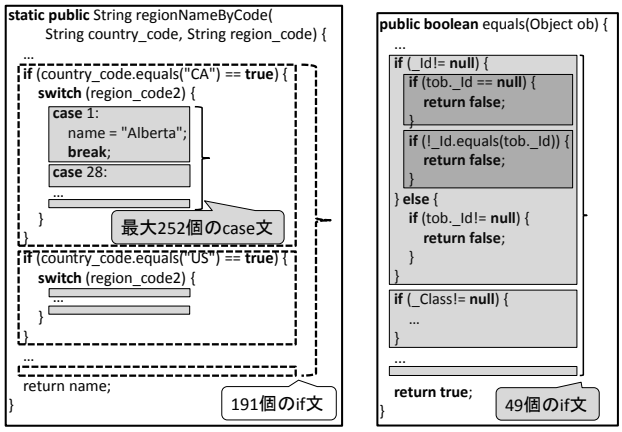


(b) 行数

図 8　閾値の推移による複雑なメソッド検出数 (被験者 B の場合)
Fig. 8　Comparison between the Ordering with Each Metric

実験結果を図 7 に示す．例えば被験者 A の場合，正解メソッドの数は 28 個である．各メトリクス名が示す値は，そのメトリクス値上位に含まれる正解メソッドの数であり，このグラフからはサイクロマチック数，行数ともに，提案手法を適用した方が被験者 A に対する正解メソッドをより多く上位に含んでいることが分かる．すなわち被験者 A にとって提案手法は有用である．全被験者の結果を見ると，サイクロマチック数の場合は 8 人中 7 人の被験者について提案手法が有用であった．一方，行数の場合，提案手法が有用であるといえるのは 8 人中 2 人の被験者についてのみで，残りの 6 人中 4 人の被験者については，提案手法適用の有無に拘らず全ての正解メソッドを上位に含んでいたため評価が行えなかった．

続いて，上位とみなす閾値を変化させて上位に含まれる正解メソッド数を計測し，より詳細な分析を行った．図 8 は，各メトリクス値上位に含まれる被験者 B の正解メソッド数を，上位とみなす閾値ごとに表したグラフである．正解メソッドのみを上位に含む場合が理想であるため，各メトリクスによる正解メソッド数の推移がグラフ中の点線で示した形に近いほど，人が感じる複雑さを表現していることになる．グラフより，CC よりも FCC の方が，また LOC よりも FLOC の方が被験者 B の感じる複雑さを表現していることが分かり，提案手法が有用で



(a) 図 4(a) に含まれるメソッド　　(b) 図 5(b) に含まれるメソッド

図 9　発見された繰り返し構造

Fig. 9　Examples of Repeated Structures

あるといえる．特にサイクロマチック数で比較した場合，CC 値の上位 10 個中には 2 個の，FCC の上位 10 個中には 9 個の正解メソッドが含まれており，従来の計測手法よりも大きく改善できたといえる．

閾値の違いによる正解メソッド数の比較を他の被験者でも行ったところ，サイクロマチック数，行数ともに，8 人中 7 人の被験者に対して提案手法が有用であることが確認できた．

5. 考察と結果の妥当性

本章では，提案手法によるメトリクス計測値の変化や計測値による順位の変化が妥当であったか考察を行う．

まず，図 4(a) に含まれるメソッドの中から，最も CC の値が高いメソッドについて調査を行った．このメソッドは，CC 値が 4,377，FCC 値が 371 で，提案手法の適用によってサイクロマチック数が大きく減少している．このメソッドは図 9(a) に示すように，1 つの if 文の中に 1 つの switch 文を含むという構造が 191 個存在し，各 switch 文中に現れる case 文が最大で 252 回繰り返されているという巨大な構造であった．提案手法では，同じ構造の繰り返しでも繰り返し回数の異なるものは類似した構造とみなしていないため，case 文は折りたたむことができても if 文を折りたたむことはできなかった．各 if 文はネスト内に含まれる case 文の繰り返し回数以外は類似していると考えられる．この例のように，case 文の繰り返し回数はあまり重要ではない可能性がある．

続いて，図 5(b) に含まれるメソッドについて調査を行った．このソフトウェア中で CC 値 100 以上のメソッドは全て，提案手法の適用によって FCC 値が 5 と計測されていた．最も高い CC 値は 199 で，図 9(b) に示すようにネスト内に if 文を 3 つ含む if 文が 49 回繰り返されたメソッドであった．また，このソフトウェアに含まれる FCC 値最大のメソッドは CC 値が 25，FCC 値が 24 で，CC 値による順位付けでは 455 番目に出現している．このメソッドは if 文や for 文による深いネスト構造を持っており，図 9(b) のメソッドと比べても非常に複雑な構造である．しかし，従来のメトリクス値による順位付けでは上位に提示することができない．従って，提案手法を適用することで複雑なメソッドの特定が容易になったといえる．

6. あ と が き

本稿では，従来のサイクロマチック数が人が感じる複雑さを表していないという問題点に着目して，プログラム中の繰り返し構造を折りたたんでメトリクスを計測する手法を提案した．大規模なオープンソースソフトウェアに対して提案手法を適用し，メソッド単位でメトリクスの計測結果を比較したところ，繰り返し構造が含まれるメソッドが多く存在し，多くのソフトウェアでその影響を受けるという結果を得た．更に，これらのメトリクスは人の主観的な複雑さを表現していることを確認した．

今後の課題は以下の 2 点である．

- 提案手法を用いた計測結果と，既存のサイクロマチック数改善手法との比較を行う．例えば pmccabe [9] は条件節中の論理演算子の数も分岐の数として含めているため，このツールの結果と比較することで，サイクロマチック数の更なる改善が見込める．
- 繰り返し構造の折りたたみ手法を応用したソフトウェア開発支援を行う．例えば Eclipse などの統合開発環境上で，ソースコード中の繰り返し構造を可視化することで，ソースコードの可読性を向上させることができると考えられる．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究 (A)(課題番号 : 21240002)，萌芽研究 (課題番号 : 23650014, 24650011)，若手研究 (A)(課題番号 : 24680002) の助成を得た．

文　　献

[1] N. Nagappan, T. Ball and A. Zeller: “Mining metrics to predict component failures”, Proc. of 28th Int. Conf. on Softw. Eng., pp. 452–461 (2006).

[2] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams and A. E. Hassan: “Revisiting common bug prediction findings using effort-aware models”, Proc. of 26th IEEE Int. Conf. on Softw. Maintenance, pp. 1–10 (2010).

[3] F. Simon, F. Steinbrückner and C. Lewerentz: “Metrics based refactoring”, Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, CSMR ’01, pp. 30–38 (2001).

[4] D. C. Atkinson and T. King: “Lightweight detection of program refactorings”, Proceedings of the 12th Asia-Pacific Software Engineering Conference, pp. 663–670 (2005).

[5] T. McCabe: “A complexity measure”, IEEE Trans. Softw. Eng, **SE-2**, 4, pp. 308–320 (1976).

[6] S. R. Chidamber and C. F. Kemerer: “A metrics suite for object oriented design”, IEEE Trans. Softw. Eng., **20**, 6, pp. 476–493 (1994).

[7] R. P. L. Buse and W. R. Weimer: “Learning a metric for code readability”, IEEE Trans. Softw. Eng, **36**, 4, pp. 546–558 (2010).

[8] A. Jbara, A. Matan and D. G. Feitelson: “High-mcc functions in the linux kernel”, Proc. of 34th Int. Conf. on Program Comprehension (2012).

[9] “pmccabe”. `http://http://manpages.ubuntu.com/manpages/lucid/man1/pmccabe.1.html`.

[10] C. Lopes, S. Bajrachaya, J. Ossher and P. Baldi: “Uci source code data sets”. `http://www.ics.uci.edu/~lopes/datasets/`.