Vol.2019-SE-202 No.13
2019/7/13

ソフトウェアクラスタリングにおける遍在モジュール特定の使用均等性計算による方式

矢野 啓介[1,a)]　松尾 昭彦[1,b)]

概要：ソフトウェアを構成するプログラム群を依存関係等によって関係の近いもの同士の集合に分割するソフトウェアクラスタリング技術はよく研究されており，ソフトウェアの実態を分析する活動への実用に供されてもいる．ソフトウェアクラスタリングの研究課題のひとつに，様々なプログラムと関係するためどのクラスタに含めることも不適当なプログラム，すなわち遍在モジュール (omnipresent module) の特定がある．あらかじめ遍在モジュールを特定し除去したうえで分割することにより，得られたクラスタがより妥当なものとなる．ところが既存の手法では，多数のプログラムから呼ばれているもののユーティリティの類いでなくシステム機能実現上の重要なプログラムを遍在モジュールとみなして除去してしまうことがある．そこで，本研究では，プログラムのメンバレベルで見てもどのプログラムからも満遍なく同じように使用されている度合い，すなわち使用均等性を定義し，他のどのプログラムとも特別な関係のないようなプログラムを遍在モジュールとみなす手法を提案する．開発手法により，従来技術において遍在モジュールの判定の不適当だったケースを改善できることが確認できた．

キーワード：ソフトウェア保守，ソフトウェアクラスタリング，遍在モジュール

1. はじめに

大規模で複雑なソフトウェアがどのように構成されているかを分析する手法として，理解や管理の容易なより小さな単位へと分割するソフトウェアクラスタリング技法が以前より研究され実用に供されてもいる．ソフトウェアは長年にわたり拡張や修正されていくことで元の設計から変わっていくことがある．そのような場合に，設計当初の意図でなく実態としてどう構成されているかを調べるために有用である．ソフトウェアのアーキテクチャを復元したり [1]，実態を可視化したりするために使われたり [2]，近年では一枚岩のシステムをマイクロサービスアーキテクチャへと移行する目的でサービス候補を見付け出すためにクラスタリング技法が用いられることもある [3][4].

ソフトウェアクラスタリングの研究課題のひとつとして，遍在モジュール (omnipresent module) の問題がある．これは，ユーティリティクラスのように，システム全体の様々な箇所と関連するプログラムであり，どのクラスタにも所属させることが不適当なものである [5][6]．遍在モジュールはノイズのように働き，クラスタリング結果の理解を難しくする要因となる．そこで従来の研究ではしばしば，クラスタリングに先立って遍在モジュールを特定し，除去したのちにクラスタリングを適用するという手法をとる．

従来の研究において，どのように遍在モジュールを特定するかは，複数の手法が提案されてきた．よく知られているものは，Bunch [5] のように，入り（ないし出）の依存関係に着目し，多数のプログラムから参照されている（ないし多数のプログラムを使用している）プログラムを遍在モジュールとみなすという方法がベースになっている．しかし，この単純な方法には，後述の事例に見るように，汎用のユーティリティクラスではなくシステムの機能の実現に重要な役割を果たすが故に多くのプログラムから参照されるものを遍在モジュールとみなして除去してしまうという課題がある．また一方で，メソッドレベルの依存関係を考慮した重みを設けることで遍在モジュールの除去を行わなくても極力良いクラスタリング結果が得られるようにした手法も提案されているが [7]，後述するように，メソッドレベルの依存関係に特徴が現れない場合は依然として他の従来手法と同じような遍在モジュールの問題が残る．

本稿において提案する手法は，クラスとメソッドのような階層関係がある場合に，メソッドレベルでみても多数のクラスから偏りなく使用されているかどうかという基準を

[1] 株式会社富士通研究所
Fujitu Laboratories Ltd., Kawasaki, Kanagawa 211–8858, Japan
a) yano@jp.fujitsu.com
b) a_matsuo@jp.fujitsu.com

設けて判定する．これにより，一部のクラスと特別な関係が認められるようなクラスは遍在モジュールとみなさないようにし，一方，メソッドレベルでも一様な使われ方をしているクラスは遍在モジュールとみなすようにすることで，上に述べた課題に対処する．

本稿の以降の部分は以下の構成をとる．まず第 2 節で関連研究を概観してそれらの特徴を述べ，続く第 3 節で開発手法を説明する．その手法を次の第 4 節で実際のソフトウェアに適用してどのように動作するかを示す．最後はまとめである．

2. 関連研究

ソフトウェアクラスタリングには様々な手法が提案されている．ソースファイル間の依存関係に基づいてまとまりを発見するものや [5]，ソースファイル中の語彙に着目して意味的に近いものを集める手法 [8]，また両者の組み合わせ [9][10] 等がある．本稿が扱う遍在モジュールの問題は依存関係に基いたクラスタリングにおいて問題となる事柄であるため，以下ではもっぱら依存関係によるクラスタリングを取り上げる．依存関係の分析には静的解析と動的解析の両方が考えられる．多くの研究は静的解析で得られる依存関係，例えばメソッド呼び出しや型の参照などを用いている．実行時のトレースのような情報を用いた動的解析の結果を併せて用いるものもあるが，その場合でも多数のクラスから使用されるクラスつまり遍在モジュールがノイズとして働く問題は共通のものといえる [11]．

前もって遍在モジュールを判定し依存関係のグラフから除去するという発想は新しいものではなく，1990 年の Muller と Uhl [12] に既に見える．この手続きが必要である理由として，遍在モジュールがシステム構造を隠してしまうためと説明されている．判定方法としては，あるノードを利用するノードの数が所定の閾値以上であれば遍在モジュールとみなすという手法がとられている．

ソフトウェアクラスタリングの研究として Mancoridis らの Bunch [5] はよく知られたものである．これは依存関係を用いて，プログラム（ソースファイル）を節点とし，ふたつのプログラム間の依存関係を辺とするグラフから，モジュール性の高い部分の集合へと全体を分割するものである．遍在モジュールを特定し除去する仕組みも含まれている．遍在モジュールの自動的な特定は，プログラムの依存関係の入次数ないし出次数が平均の例えば 3 倍といった閾値を設定してそれ以上の次数を持つプログラムを遍在モジュールとみなすというように行なわれる．この手法は他の著者によるソフトウェアクラスタリングの研究でも採用されている [13]．

Wen と Tzerpos [6] は，接続されているプログラムではなくサブシステムの数に基いて遍在モジュールの判定を行う方法を提案している．これは様々な機能から横断的に使われているという意味を表す有用な方法と考えられる．ただ，サブシステムかソースファイルかという違いはあるにせよ，多数のモジュールと接続されるかどうかによって判定しているという面では同じである．

一方, SArF [7] はクラス間の依存関係に Dedication Score（以下，日本語では専念度スコアと呼ぶ）を定義して依存関係グラフの辺の重みとしてクラスタリングに用いることで，遍在モジュール除去のステップを不要としている．専念度スコアは，あるクラスに注目したとき，多数のクラスから使用されているならば小さく，逆に特定のクラスからのみ使われているときは大きくなるよう計算される．これにより，ログ出力クラスのように多数のクラスから使用されているものは各クラスからの重みが小さくなる．また，クラスだけでなくメソッドレベルの依存関係も考慮する．これにより，ログ出力クラスの中のログ出力メソッドのみを使用している大多数の使用クラスからの依存関係は重みが小さくなるのに対して，ログ機能の初期化のような他と異なるメソッドを使用している少数のクラスについては重みが大きくなるという計算が行なわれ，クラスタリングにおいてその特定少数のクラスと一緒のクラスタに分類されやすくなる．この性質により，多数のクラスから使用されるクラスであっても，極力適切なクラスタを見付けてそこに所属させることができる．

しかし，SArF を用いてもなお不適当な結果が生じることはある．SArF は他と依存関係のあるクラスを必ずどこかのクラスタに分類するため，メソッドレベルの依存関係を分析しても使用のされ方に全く偏りのない場合は，たまたまどこかの機能のクラスタに所属させられることになる．このようなクラスはそのクラスタに所属する理由を見出すことが困難であり，ノイズのように働くという遍在モジュールの問題が依然として発生する．このような課題の具体的な例はのちに第 4 節の事例において取り上げる．

3. 提案手法

本稿において提案する遍在モジュール特定手法は，SArF の手法に基いてクラス間の専念度スコアを計算したのちに，その値を利用して，クラスが他の多数のクラスからメソッドレベルでみてもどのクラスからも偏りなく同じように使用されているかどうかを判定する．どのクラスからも同じように使用されているとみなされたクラスは，遍在モジュールとみなしてクラスタリングから除去する．この，他の多数のクラスから同じように偏りなく使用されている度合いを，本稿では使用均等性と呼ぶ．つまり，使用均等性の高いものを遍在モジュールとみなしてクラスタリングから除去するという方法である．逆に，あるクラスからだけは他と異なるメソッドが利用されているというような特別な関係が存在するときには，使用均等性が低いとみなす．他と異なる特別な関係を持つクラスとのつながりが強いと

解釈できるため，クラスタリングからは除去しない．

あるクラスの使用均等性を本稿では以下のように定義する．

ある依存関係の専念度スコアは，Kobayashi ら [7] の式 (2) より下記の $D_M(A,B)$ として求められる．注目する依存関係の依存元クラスを A，依存先を B とするとき，この依存関係 (A,B) の専念度スコアが $D_M(A,B)$ である．

$$D_M(A,B) = \sum_{m \in M_{AB}} \frac{1}{\mathrm{xm}_{fanin}(m) \cdot \mathrm{mx}(B)} \tag{1}$$

ここで，M_{AB} は A のメンバ（メソッド，フィールド）に依存されている B のメンバの集合であり，$\mathrm{xm}_{fanin}(m)$ はメンバ m に依存する B の外部のメンバの数である．$\mathrm{mx}(B)$ は外部のメンバに依存されている B のメンバの数である．

これを用いて，B の使用均等性を以下のように定義する．A は B への依存関係を持つクラスである．

$$e = 1 - \max_A \{1 - D_M(A,B)\} \tag{2}$$

e の値が所定の閾値（例えば 0.85）未満ならば，B は遍在モジュールではないとする．

一方，e の値が閾値以上ならば，多重度 c を以下により求める．

$$c = 1 - \frac{1}{\mathrm{fanin}(B)} \tag{3}$$

ここで，$\mathrm{fanin}(B)$ は，B への依存関係を持つクラスの数である．c に閾値（例えば 0.9）を設けて，e だけでなく c もまた閾値以上ならば遍在モジュールであると判定する．

使用均等性 e が高いことは，専念度スコアの高い依存元クラスがいないということであり，したがって特定クラスとの結び付きが弱いことを意味する．この専念度スコアはクラスとメソッドの階層関係が考慮された計算であることから，e にはメソッドレベルでどのクラスからも同じように使われているかどうかが加味されることになる．その際に，多重度 c が高く，少数ではなく多数のクラスから使用されているほど，遍在モジュールとしての性質は高いと考えられる．つまり，使用均等性 e と，多重度 c との組み合わせによって，特定クラスと特別な結び付きがなくかつ多数のクラスから使われていることを意味している．

上に見た判定方法を用いて，クラスタリング全体の処理の流れは以下のようになる．

(1) クラス間の依存関係を解析してグラフ構造を得る
(2) 各依存関係に対して専念度スコアを計算する
(3) 専念度スコアに基いて各クラスの使用均等性および多重度を求めて遍在モジュールを特定する
(4) 特定された遍在モジュールをグラフ構造から除去してクラスタリングを行う

このうち 1 番目は一般的な静的解析の技法を用いて行われる．2 番目は SArF の手法を用いる．3 番目が本稿に特有な部分であり，上に説明した手法である．4 番目は遍在モジュールを除去するソフトウェアクラスタリング技法において一般的な手順である．

なお，ここではオブジェクト指向言語を念頭においてクラスとメソッドという呼び方をしているが，他の種類のプログラミング言語であっても類似の階層関係がプログラム構成にある場合には読み替えて適用可能と考えられる．例えば C 言語ならばソースファイルと関数を対応させることがあり得る．

表 1 クラス ProcessViewerResource に依存するクラスとその専念度スコア

Table 1 Classes depending on the class ProcessViewerResource and their dedication scores

クラス名	スコア
BusinessProcessFrame	0.036
ReportDialog	0.036
ViewerUtils	0.029
(62 クラス 略)	
AlternateViewModel	0.008
DetailDataModel	0.008

4. 事例

本節では，前節の提案手法がどのように動作するかを事例を通じて示す．

事例として用いる分析対象ソフトウェアは，ある企業で使われている業務プロセスビューアである．Java 言語で書かれており，GUI にて業務プロセスを表示することが主な機能である．このシステムの中のいくつかのクラスを判定事例として用いる．なお，機密保持上の理由により，クラス名等の情報は要点の理解に支障のない程度に実際のものから変更している．

以下の小節では，遍在モジュール判定の例を 3 例挙げて従来技術と比較する．比較する従来技術としては，Bunch の遍在モジュール判定の方法，つまり入次数の平均の 3 倍を閾値として，それより大きければ遍在モジュールとみなす方法をとる．最後に，クラスタリング結果の利用例として著者らが想定している可視化の例を掲げ，遍在モジュールがどのように処理されるかを示す．

4.1 従来技術同様に適切に遍在モジュールとして判定される例

最初の例は，従来技術でも提案手法でもともに遍在モジュールとして判定され，その結果が妥当であるようなクラス ProcessViewerResource の例である．

表 1 は，クラス ProcessViewerResource を参照するクラスと，その参照元クラスから当該クラスへの依存関係の専念度スコアを，スコアの降順に示している．ただし，全部で 67 クラスと数が多いため，途中を省略している．

表 2 クラス BusinessProcessModel に依存するクラスとその専念度スコア
Table 2 Classes depending on the class BusinessProcessModel and their dedication scores

クラス名	スコア
BusinessProcessFrame	0.151
ImageFileWriter	0.100
ReportDialog	0.050
(77 クラス 略)	
DetailListFrame	<0.001
SortFrame	<0.001

このクラスの使用均等性を計算する．まず，このクラスへの入りの依存関係の専念度スコアの最大値は 0.036 である．したがって使用均等性は $e = 1 - 0.036 = 0.964$ と求められる．この値は閾値 0.85 より大きい．したがってこのクラスは遍在モジュールである可能性がある．そこで式 (3) を計算すると，$c = 1 - 1/67 = 0.985$ となる．この値は閾値 0.9 より大きいので，このクラスは遍在モジュールとみなされる．

従来技術では，このソフトウェアの入次数の平均の 3 倍が約 12.9 であり，これを閾値とすると当該クラスは入次数が 67 であることから閾値より大きく，したがって従来技術によっても遍在モジュールとみなされる．

つまり，このクラスは従来技術によっても提案手法によっても遍在モジュールと判定される．このクラスは国際化文字列リソースを提供するクラスであり，システム上の特定の機能に属するものではなく様々な機能から使用される．したがって，これを遍在モジュールとみなすことは適当である．従来，SArF の手法では，このクラスも何らかのクラスタに所属させられるが，当該クラスタのクラスと特に結び付きが強いことによってではないため，なぜこのクラスがほかではなくそのクラスタに入っているのかを理解することが困難である．こうしたクラスは，SArF 以外の手法と同じように，クラスタリングから除外するのが適当であるといえる．

4.2 従来技術では不適当に遍在モジュールとしてしまう例

次に，従来技術では不適当に遍在モジュールとして判定してしまう例を挙げ，提案手法では遍在モジュールとはみなさないことを示す．

クラス BusinessProcessModel は 82 クラスと多数のクラスから参照されるので，従来技術では閾値を大きく上回り，遍在モジュールとみなされてクラスタリングから除外される．しかし実際は，このプロセス分析システムの中心的な機能において重要な役割を果たすモデルクラスであり，ユーティリティクラスではない．したがって従来技術の判定結果は不適当である．

提案手法によって遍在モジュールか否かを判定する．表 2 にこのクラスへの依存関係を持つクラスとその依存関係の専念度スコアを示す．ただし数が多いため途中を省略している．専念度スコアの最大値は 0.151 であり，使用均等性は $e = 1 - 0.151 = 0.849$ となる．この値は閾値に満たない．このため，本クラスは遍在モジュールとはみなさず，クラスタリングに含まれる．クラスタリングの際は専念度スコア上位のクラスと一緒のクラスタに分類されることが期待される．

表 3 クラス StreamCloser に依存するクラスとその専念度スコア
Table 3 Classes depending on the class StreamCloser and their dedication scores

クラス名	スコア
ViewerUtils	0.112
GraphModel	0.112
ReportManager	0.112
(6 クラス 略)	
ChartFrame	0.079
OutputListener	0.079

この例では，依存元クラスごとに専念度スコアに違いがあることが結果につながっている．つまり，多数のクラスから使われているとしても，その使い方がどれも同じようではなく，あるクラスは他のクラスとは異なるメソッド呼び出しをしていることによる．より具体的には，BusinessProcessFrame はプロセス表示のためのクラスであり，モデルクラス BusinessProcessModel の機能設定を含め様々なメソッドを使用する．一方，それ以外のクラスが使用するのはモデルの情報を得る等の一部のメソッドに限定されているという違いがある．

4.3 従来技術では除去されずに見過ごされる例

次に，前の例とは逆に，従来技術では遍在モジュールとみなされなかったが，実際には遍在モジュールとして働いており，提案手法で検出できた例を示す．

クラス StreamCloser は 11 クラスから使用されている．この数は従来技術では閾値 (12.9) 未満のため遍在モジュールとみなされなかった．しかし実際は，このクラスは入出力処理を簡便に記述するためのユーティリティクラスであり，様々な機能から利用されているもので，特定機能には属さない．したがって，従来技術の判定結果は不適当である．

提案手法によってこのクラスが遍在モジュールか否かを判定する．表 3 にこのクラスへの依存関係を持つクラスとその依存関係の専念度スコアを示す．このクラスへの入りの依存関係の専念度スコアの最大値は 0.112 であり，使用均等性は $e = 1 - 0.112 = 0.888$ と求められる．閾値 0.85 より大きいので，多重度を計算すると，$c = 1 - 1/11 = 0.909$. これも閾値 0.9 より大きいため，遍在モジュールとみなされる．つまり，従来技術では不適当であった判定結果が改

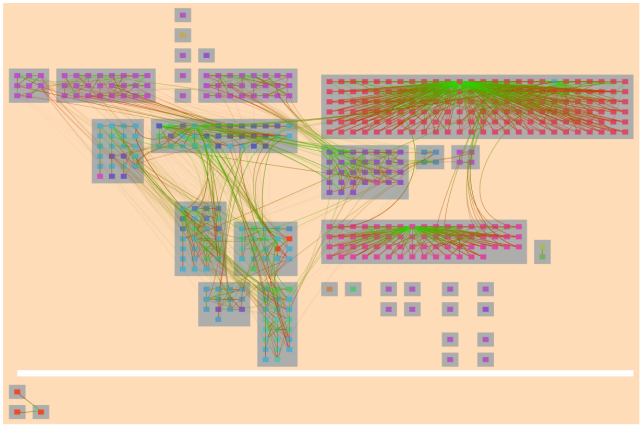

図 1 遍在モジュールを除去してクラスタリングした結果の可視化
Fig. 1 Visualization of the resulting clusters removing omnipresent modules

図 2 遍在モジュールとの依存関係の可視化
Fig. 2 Visualization of the dependencies between omnipresent modules and nonomnipresent modules

められ，特定機能に属さないクラスがクラスタリングから除外されるようになった．

この例は，呼び出し元のクラスが複数あってもどれも決まりきったメソッド呼び出しをしており，呼び出し元による違いがほとんどないことから，使用均等性が高い結果になっている．

4.4 可視化

著者らの想定しているクラスタリングの利用法として，クラスタリング結果を用いたソフトウェア構造の可視化がある．著者らの以前の研究である SArF Map [2] は，都市の地図を模したソフトウェア可視化手法である．得られたクラスタを 2 次元平面上に配置し，クラスを建物，クラスの集まったクラスタを街区として表現する．建物の色はクラスの属する Java パッケージ（他の言語ではソースファイルのディレクトリ等，類似の単位）によって割り当てられる．ソフトウェアの都市メタファーによる可視化としてよく知られている CodeCity [14] はパッケージ構造に従って配置するのに対し，SArF Map はクラスタリング結果に従うことから，設計者の与えた分類ではなくソフトウェアの実際の依存関係に従った構造が表現される．街区として描画されるクラスタはソフトウェアの何らかの機能を実装していることが期待される [7]．ソフトウェアクラスタリング結果のこうした可視化手法は，企業のソフトウェア開発活動において品質の分析に活用されている実績もある [15]．

図 1 は，本稿の手法を用いて特定された遍在モジュールを取り除いたうえでクラスタリングを実施し，得られたクラスタを SArF Map の手法によって街区として表現したものである．本可視化手法は Z 軸方向にプログラムメトリクスを割り当てた 3 次元表現が可能であるが，今回はメトリクスを用いないので上面図によって表現している．灰色の長方形で描画される街区がクラスタに対応し，その中の色のついた正方形がクラスである．各クラス（建物）の間には，依存関係を示す曲線を，緑（依存元）から赤（依存先）へのグラデーションによって描画している．本来はクラスタの理解の手がかりのためにソースファイルから得たクラスタの特徴語を街区に重ね合わせて表示することもできるが [16]，この図では省略している．図の下部には，除去された遍在モジュールを別立てとして配置している．これにより，業務機能的なまとまりとなることが期待される街区の中にその機能と直接関係のないユーティリティ的なクラスが紛れ込むことを防ぐとともに，除去されたクラスの存在が認識できる．可視化ツールの内部に保持しているデータとしては遍在モジュールと他のクラスとの間の依存関係も持っているが図 1 には描画していない．

一方，図 2 は，同じデータを用いて，遍在モジュールと他のクラスの間の依存関係の曲線を描画した結果である．多数のクラスに接続していることがわかる．遍在モジュールとして除去されたクラスが実際にどのクラスと関係があるかを知りたいときにはこのように表示を切り替えて調べることができる．

こうした可視化を行う場合，遍在モジュールとして除去される（すなわち，図において下部に配置される）ものが多すぎると，クラスのまとまりを視覚的に表現するという可視化の意義が損なわれるおそれがある．この観点からは，除去される遍在モジュールは控え目に判定されることが望ましい．今回の例では，提案手法が遍在モジュールと判定したのは図にあるように 3 クラスのみだった．内訳は，前記 4.1 節のリソース管理クラスと 4.3 節の StreamCloser，および static メソッドを集めたユーティリティクラスであり，いずれも遍在モジュールとみなすのが妥当なものである．それに対し，従来技術のように入次数の大小だけで判定すると多数のクラスが遍在モジュールとみなされて図の下部に並ぶことになる．今回従来技術で判定されたのは 23 クラスあった．その中には上記 4.2 節の例において述べたような当該システムの機能上で重要な役割を果たすモデル

クラスも含まれている．そうしたクラスはクラスタリング対象として，図の本体部分に配置された方が良い．他の用途には異なる方針（たとえば，多めに遍在モジュール扱いした方がよい）が適している可能性もあり得るが，本稿が想定しているシステムの現状を分析するために可視化するという目的には，除去しすぎない判定の方が適していると考えられる．

5. おわりに

本稿では，ソフトウェアクラスタリングにおける遍在モジュール特定の新しい手法を提案し，実際のソフトウェアに適用して，どのように動作し，従来技術で不適当な結果となる例に対して改善するかを説明した．本手法はメソッドレベルでみてどのクラスからも同じように偏りなく使われているかどうか，すなわち使用均等性を考慮に入れることで，クラス間に他と異なる特別な関係がないかどうかを評価している．クラス間の依存関係の数の大小のみで判定する従来手法と異なり，システム機能の実現上で重要な役割を担うクラスを避け，ユーティリティ的な性質の強いものが遍在モジュールとして判定されやすくなることを例を通じて示した．

今回は使用均等性という概念に対して比較的単純な定義を与えて計算しているが，より精緻化した計算方法を定義し実験して効果を検証することが今後の発展として考えられる．それらの手法と従来手法とをあわせて，複数の例を用いて統計的な評価を行うことが課題として挙げられる．

参考文献

[1] Tzerpos, V. and Holt, R. C.: ACDC: An Algorithm for Comprehension-Driven Clustering., *WCRE*, pp. 258–267 (2000).

[2] Kobayashi, K., Kamimura, M., Yano, K., Kato, K. and Matsuo, A.: SArF Map: Visualizing software architecture from feature and layer viewpoints, *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pp. 43–52 (online), DOI: 10.1109/ICPC.2013.6613832 (2013).

[3] Mazlami, G., Cito, J. and Leitner, P.: Extraction of Microservices from Monolithic Software Architectures, *2017 IEEE 24th International Conference on Web Services*, pp. 524–531 (online), DOI: 10.1109/ICWS.2017.61 (2017).

[4] Kamimura, M., Yano, K., Hatano, T. and Matsuo, A.: Extracting Candidates of Microservices from Monolithic Application Code, *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 571–580 (online), DOI: 10.1109/APSEC.2018.00072 (2018).

[5] Mancoridis, S., Mitchell, B. S., Chen, Y. and Gansner, E. R.: Bunch: a clustering tool for the recovery and maintenance of software system structures, *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pp. 50–59 (online), DOI: 10.1109/ICSM.1999.792498 (1999).

[6] Wen, Z. and Tzerpos, V.: Software clustering based on omnipresent object detection, *Proceedings - IEEE Workshop on Program Comprehension*, pp. 269–278 (online), DOI: 10.1109/WPC.2005.31 (2005).

[7] Kobayashi, K., Kamimura, M., Kato, K., Yano, K. and Matsuo, A.: Feature-gathering dependency-based software clustering using Dedication and Modularity, *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 462–471 (online), DOI: 10.1109/ICSM.2012.6405308 (2012).

[8] Kuhn, A., Ducasse, S. and Gîrba, T.: Semantic clustering: Identifying topics in source code, *Information and Software Technology*, Vol. 49, No. 3, pp. 230–243 (2007).

[9] Scanniello, G., D'Amico, A., D'Amico, C. and D'Amico, T.: Using the Kleinberg algorithm and vector space model for software system clustering, *IEEE International Conference on Program Comprehension*, pp. 180–189 (online), DOI: 10.1109/ICPC.2010.17 (2010).

[10] 矢野啓介，松尾昭彦：依存関係に基づいたソフトウェアクラスタの意味的凝集度を用いた調整，電子情報通信学会技術研究報告 vol. 118, no. 69，pp. 43–48 (2018).

[11] Patel, C., Hamou-Lhadj, A. and Rilling, J.: Software Clustering Using Dynamic Analysis and Static Dependencies, *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pp. 27–36 (online), DOI: 10.1109/CSMR.2009.62 (2009).

[12] Muller, H. and Uhl, J.: Composing subsystem structures using (k,2)-partite graphs, *Proceedings. Conference on Software Maintenance 1990*, IEEE Comput. Soc. Press, pp. 12–19 (online), DOI: 10.1109/ICSM.1990.131315 (1990).

[13] Paixao, M., Harman, M. and Zhang, Y.: Multi-objective Module Clustering for Kate, *International Symposium on Search Based Software Engineering*, pp. 282–288 (2015).

[14] Wettel, R. and Lanza, M.: Visualizing software systems as cities, *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 92–99 (online), DOI: 10.1109/VISSOF.2007.4290706 (2007).

[15] 三神郷子，中嶋久彰：メトリクスを用いてネットワークソフトウェアの内部品質を可視化する技術，電子情報通信学会論文誌，Vol. J100-B, No. 5, pp. 356–364（オンライン），DOI: 10.14923/transcomj.2016NSI0001 (2017).

[16] Yano, K. and Matsuo, A.: Labeling Feature-Oriented Software Clusters for Software Visualization Application, *2015 Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, pp. 354–361 (2015).

正誤表

下記の箇所に誤りがございました．お詫びして訂正いたします．

訂正箇所	誤	正
3 ページ 式(2)	$e = 1 - \max_A\{1 - D_M(A, B)\}$	$e = 1 - \max_A D_M(A, B)$