社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－SE－159 (16)
2008／3／18

# ソフトウェアメトリクスの統合によるソースコード変更の選択

佐々木 祐輔† 林 晋平† 佐伯元司†

ソースコードに対する変更として適切なものを選択することは難しい．本稿では，ソフトウェア開発プロジェクトの方針に基づいて，各開発者が適切な変更を選択することを支援する手法を提案する．提案手法では，開発者個人の主観による影響を抑制するために，複数のソフトウェアメトリクスを統合した評価関数によって変更の選択肢の優劣を判断する．また，プロジェクトの方針に基づいた選択を実現するために，ソースコードに対する変更の選択を，複数のメトリクスを評価項目とする多目的意思決定とみなすことにより，評価関数の作成を階層分析法を応用した方法によって行う．本稿で行った評価では，提案手法は変更の選択の支援に有用であるという結論を得た．

# Selecting Alternatives of Source Code Modification with Integrating Software Metrics

Yusuke Sasaki,† Shinpei Hayashi† and Motoshi Saeki†

Selecting the most appropriate alternatives of source code modification is difficult. This paper proposes a technique to help have each developer of software development project selecting appropriate modifications based on the project's commitment. In the technique, we judge the order of superiority of alternative modifications by creating an evaluation function with integrating multiple software metrics to suppress the influence of each developer's subjectivity. Considering selecting an alternative of source code modification as a multiple criteria decision making, we create the function with Analytic Hierarchy Process. An evaluation shows the efficiency of the technique.

## 1. はじめに

ソフトウェアの保守性を向上させるためにリファクタリング[1]などの変更が行われるが，その効果を評価し，適切な変更を選択することは難しい．その原因として，保守性を直接計測できないことが挙げられる．保守性を推定する手段として，ソフトウェアメトリクスがある．ただし，一つのメトリクスの値が表しているのはソフトウェアの性質の一つであり，保守性を総合的に評価するためには多くのメトリクスを調べる必要がある．評価に使用したメトリクスの種類毎に変更の選択肢の優劣が異なり，各メトリクスをどれだけ重視するかによって，優れていると判断される選択肢が異なる場合がある．そのため，選択肢の優劣は一意に評価できない．この優劣は開発プロジェクトの方針に基づいて判断すべきであるが，選択を行う度に合意形成を行うことは非効率的である．

† 東京工業大学 大学院情報理工学研究科 計算工学専攻
Department of Computer Science, Graduate school of Information Science and Engineering, Tokyo Institute of Technology

本稿では，メトリクス間にトレードオフがある状況に対応するために，多目的意思決定の手法を用いて設定した判断基準によって，ソースコードの変更を自動的に選択する手法を提案する．提案手法では，複数のメトリクスの値をパラメータとする評価関数の値によって変更の選択肢の優劣を判断する．複数のメトリクスを評価関数で統合することにより，保守性の総合的な評価に基づき，かつ開発者に依存しない判断を実現する．また，ソースコードの変更の選択を，複数のメトリクスを評価項目とする多目的意思決定とみなし，階層分析法[2]を応用して評価関数を作成する手法をとる．提案手法の実現可能性および有用性を確認するために，手法の実施を通して評価を行う．

本稿の以降の構成を示す．2節では，保守性を向上するためのソースコードに対する適切な変更を選択することに関する問題について述べる．3節では，本稿で提案するソースコード変更の選択の手法を説明する．4節では，提案手法の評価のために行った実験について述べる．5節では，提案手法で使用するメトリクスに関する議論を行う．6節では，関連研究を紹介する．最後に7節で本稿についてまとめ，今後の課題につい

て述べる．

## 2. ソースコード変更の選択における問題

保守性向上のためのソースコードに対する変更として適切なものを選択することは難しい．保守性は直接計測できないため，変更による保守性の変化に対する評価は，それを行っている作業者の主観に依存する．そのため，最適だと判断して実施した変更が，他の開発者（開発がしばらく進んだ後の作業者自身をも含む）には望まれていない可能性がある．しかし，変更を検討する度に，プロジェクトのリーダに判断をさせたり，開発プロジェクトの合議で判断したりすることは非効率的である．そのため，保守性の向上を確実に行うために，選択の方法をプロジェクトの開発者の間で統一し，作業者の主観に依存しない判断をすることが望まれる．

保守性向上のためのソースコード変更の例を示す．図1はリファクタリングの一種であるメソッドの抽出を行う前後のソースコードの例である．メソッドの抽出は，メソッド内の処理の一部を抜き出して別のメソッドを作成し，抜き出した元の部分では作成したメソッドを呼ぶように変更することにより，メソッドの規模の縮小を実現する．ただし，メソッドの規模以外の側面にも影響が生じる．例えば，新たなメソッドが増え，ソースコード全体の規模が大きくなるという影響がある．従って，変更を望むか否かが開発者によって異なる可能性がある．

ソースコード変更の選択に利用する保守性の評価指標として，ソフトウェアメトリクスが有用であると考える．ソフトウェアメトリクスは，ソースコードの規模や複雑さなどに関する尺度であり，その値から保守性について推定することができる．また，メトリクスの値は，機械的な計測によって得られる客観的で定量的な情報である．従って，ソースコード変更の選択の際に，メトリクスの値を作業者に依存しない判断の根拠とすることができる．図1の例において，メソッドの規模が改善されたことはメソッドの文の（NOS: Number Of Statement）の減少から，ソースコード全体の規模が悪化したことはクラスの行数（LOC: Lines Of Code）の増加から評価することができる．

ただし，メトリクスを計測するだけでは，変更の選択肢の優劣は一意に評価できない．一つのメトリクスの値が表しているのはソフトウェアの性質の一つであり，保守性を総合的に評価するためには多くのメトリクスを調べる必要がある．評価に使用したメトリクスの種類毎に変更の選択肢の優劣が異なり，各メトリクスをどれだけ重視するかによって優れていると判断される選択肢が異なる場合がある．図1の変更を適用すべきか否かの判断は，NOS と LOC およびその他のメトリクスをどれだけ重視するかに依存する．

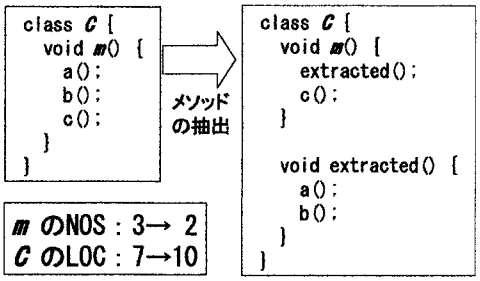


**図1** 保守性向上のためのソースコード変更の例

従って，メトリクスの計測だけでは，保守性向上のための変更作業を円滑に進めることはできない．メトリクスの間にトレードオフがある場合でも，変更の選択肢に対する優劣について，作業者の個人的な主観からの影響を受けず，プロジェクトの方針を反映した評価を行う方法が必要である．

## 3. ソースコード変更の選択の手法

### 3.1 概　　要

本稿では，ソフトウェア開発プロジェクトの方針に基づいて作成した評価関数を用いて，ソースコードに対する変更の選択を自動的に行う手法を提案する．提案手法では，ソースコードに対する変更の選択を，ソフトウェアメトリクスを評価項目とする多目的意思決定とみなす．そうすることにより，メトリクスの間にトレードオフがある状況において，変更の選択を多目的意思決定の手法を応用して支援することができる．プロジェクトの方針の反映と作業者への依存の排除を達成するために，人間の価値判断を反映した意思決定を支援する手法である階層分析法を応用して判断基準を設定することにより，変更の選択を自動化する．

提案手法を利用するプロセスは，図2のように，評価関数の作成とソースコードの変更の選択の二つの部分からなる．評価関数の作成は，プロジェクトの方針を反映した判断を実現するために，プロジェクトのリーダが参加する合議などによって行う．一方，ソースコードの変更の選択は，作業者の主観の影響を受けずに効率的に行うために，評価関数の値を用いて機械的に行う．

評価関数の作成は，基本的にプロジェクトの開始時に一度だけ行う．まず，評価項目とするメトリクスの

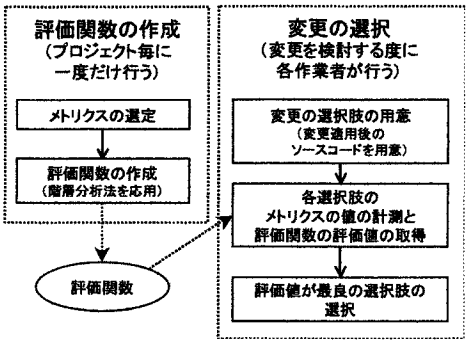


**図 2** 提案手法の利用プロセス

セットを選定する．続いて，選定したメトリクスのセットをパラメータとする評価関数の作成を，階層分析法を応用した手法によって行う．作成した評価関数は，プロジェクトの方針に変化が生じることなどが無い限り繰り返し利用できるので，この作業をリーダが参加する合議などで行うことは，作業効率の悪化をもたらさないと考える．

ソフトウェアの開発中に，保守性の向上のための変更を行うことを検討する度に，各開発者が本手法に従った作業によって判断を行う．まず，適用する候補として考えている全ての変更に対し，それぞれを適用したソースコードを用意する．現在のままのソースコードも，用意したソースコードと同列に扱う．続いて，それぞれのソースコードのメトリクスの値を計測し，作成した評価関数の値を得る．評価関数の値が最良である選択肢を採用する．用意した選択肢に対する優劣の判断が作業者に依存しないため，この作業は各開発者が個人で行うことができる．

### 3.2 階層分析法の応用

提案手法では，階層分析法の手順に従い，ソースコードに対する変更の選択を図 3 のような階層構造として捉える．階層構造の最上層（レベル 1）は最終目標である保守性の向上である．最下層（レベル 3）は最終目標のために選択できる代替案を配置する階層であり，提案手法ではソースコードに対する変更の選択肢である．なお，ここでは変更しないことも選択肢の一つとする．これら二つの階層の間に，最終目標に対する各代替案の価値を評価するための評価項目を配置する階層（レベル 2）があり，提案手法ではここには評価関数のパラメータとするメトリクスが該当する．

階層分析法では，構築した階層構造の最上層以外の各要素の，一つ上の階層の要素に対する重要度を定め，それらから決定した各代替案の評価値によって，代替

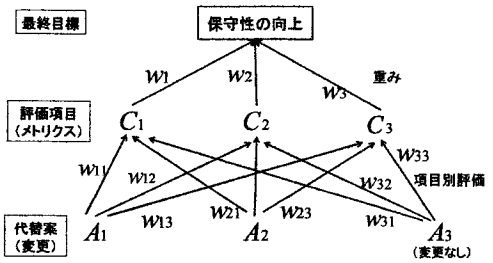


**図 3** 構築する階層構造

案を選択する．提案手法のような 3 階層の場合には，最終目標に対する各評価項目の重みと，各代替案の評価項目別の重要度を定める．$i$ 番目の代替案の評価値 $p_i$ は，評価項目の数を $n$ とし，$j$ 番目の評価項目の重みを $w_j$ とし，$j$ 番目の評価項目に対する $i$ 番目の代替案の重要度を $w_{ji}$ とすると，以下の式で表される．

$$p_i = \sum_{j=1}^{n} w_j w_{ji}. \qquad (1)$$

評価値が最も高い代替案を，最良の選択肢であると判断する．

提案手法では，メトリクスの値をパラメータとして受け取り，それらの加重平均の値を返す評価関数によって，変更の選択肢の優劣を判断する．通常の階層分析法では，階層構造の各要素の重要度は全て，後述する一対比較行列を用いた手法により手作業で決定する．しかし，提案手法では，メトリクスの重みのみを手作業で決定し，変更の選択肢のメトリクス別の重要度は，その変更を適用したソースコードの各メトリクスの値とする．メトリクスの選定と各メトリクスに対する重みの決定をプロジェクトの合議などで行うことにより，プロジェクトの方針を判断に反映させる．一方，変更の選択肢毎の評価はメトリクスの値によって行うことにより，変更を検討する度に手作業が必要にならないようにする．

メトリクスの重みの決定は，一対比較行列を用いた手法によって行う．一対比較行列は比較する要素の数だけの行及び列を持つ正方行列である．図 3 の階層構造のメトリクスの重みを決定するための行列を図 4 に示す．

この行列の各成分の値は，要素を二つ取り出して比較することによって決定する．提案手法では，例えば図 4 の行列であれば，メトリクス $C_i$ の値が 1 ポイント変化することの影響力が，メトリクス $C_j$ の値が $\alpha$ ポイント変化することの影響力と同程度と考えたら，行列の $(i,j)$ 成分に $\alpha$ を，$(j,i)$ 成分に $1/\alpha$ を入れる．

$$\begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \end{array} \begin{array}{c} \begin{array}{ccc} C_1 & C_2 & C_3 \end{array} \\ \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \end{array}$$

**図 4** 一対比較行列の例

対角成分には 1 を入れる．

各要素の重要度は，作成した行列を利用した計算によって決定する．重要度の計算には様々な方法があるが，提案手法では幾何平均法[2] を使用する．幾何平均法では，一対比較行列の各行の幾何平均を重要度の基になる値とし，その値の合計で割ることで重要度の合計が 1 になるように正規化したものを算出結果とする．$n$ 個の要素を比較した行列 $A = [a_{ij}]$ から $x$ 番目の要素の重要度 $w_x$ $(1 \leq x \leq n)$ を算出する式は以下のようになる．

$$w_x = \frac{\sqrt[n]{\Pi_{j=1}^{n} a_{xj}}}{\sum_{i=1}^{n} \sqrt[n]{\Pi_{j=1}^{n} a_{ij}}}. \quad (2)$$

一対比較行列の整合性を確かめるために，C.I.（Consistency Index）を算出する．C.I. は以下の計算で求められる．

$$\text{C.I.} = \frac{\tilde{\lambda} - n}{n - 1}. \quad (3)$$

ここで，

$$\lambda_x = \frac{\sum_{i=1}^{n} (a_{xi} w_i)}{w_x}, \tilde{\lambda} = \frac{\sum_{i=1}^{n} \lambda_i}{n}. \quad (4)$$

一対比較行列の整合性が良いほど C.I. の値は小さくなり，整合性が完全に取れている場合には 0 となる．C.I. $\geq 0.1$ の場合には，一対比較の値について再検討すべきものとする．

### 3.3 メトリクスの条件

有用な評価関数を作成するためには，パラメータにするメトリクスのセットを適切なものにする必要がある．品質に影響を及ぼす，ソースコードの様々な性質を捕捉できるように，多様なメトリクスから構成すべきである．

使用するメトリクスについて，値が大きいことと小さいことのどちらが望ましいのかを統一する必要がある．ソースコードの規模を直接的に計測するメトリクスや，オブジェクト指向メトリクスとして広く利用されている C&K メトリクス[3] が小さいことが望ましいメトリクスであることから，値が小さい方が望ましいとすることへの統一の方がわかりやすい状況が多いと想定される．値が小さいことが望ましいように統一した場合，評価関数も小さい方が望ましいものとなる．

保守性の評価に必要な性質を捕捉するためには，計測の対象がソースコード内のモジュールであるメトリクスを利用することが必要になる．しかし，メトリクスに対する重み付けは人手で，選択肢を得る前に行う．そのため，数が膨大であり，かつ作成および削除される可能性があるモジュールに付随する値をそのまま評価項目にすることはできない．従って，モジュール毎の値を統合してソフトウェア全体で一つの値にして評価項目とする必要がある．

## 4. 事例による評価

### 4.1 目　　的

評価の目的は，以下の二点の確認である．

- 提案手法の実現可能性
- 提案手法を用いた判断が開発プロジェクトの方針を反映できるかどうか

まず，提案手法で実際に評価関数を作成できることを確認する．評価関数の作成が可能なのであれば，それを使用することで，作業者に依存しない判断を行えることは自明である．しかし，その評価関数による判断が，開発プロジェクトの方針を反映したものになるかどうかは，明らかではない．人間が常に自身の方針に沿った判断を正確に下せるという仮定のもとでは，人間の判断と評価関数による判断とで結果が常に一致するのであれば，その評価関数はその人間の方針を完全に反映したものであるといえる．このことから，評価関数の作成者の判断と，作成した評価関数による判断とで結果が一致するかを調べることで，作成した評価関数の性能を評価することにする．また，結果が一致する割合を見るだけでなく，結果が一致する状況と一致しない状況について分析する．それによって，提案手法がどのような場合に有用になるのか，また有用でない状況があれば，その原因と改善のための要件を考察する．

### 4.2 評価のプロセス

ソースコードに対する変更の選択肢として同じ組み合わせが与えられた状況での，評価関数による判断とその関数の作成者による判断を比較するためのプロセスを実施した．

まず予め，評価関数のパラメータとするメトリクスのセットを用意した．また，二つのソースコードから保守性が優れていると考えられるものを選ぶ問題のセットを二グループ用意した．これらの問題のソースコードの組は全て，同じソフトウェアで部分的に異なるものの組み合わせであり，ソースコード変更の前後に相

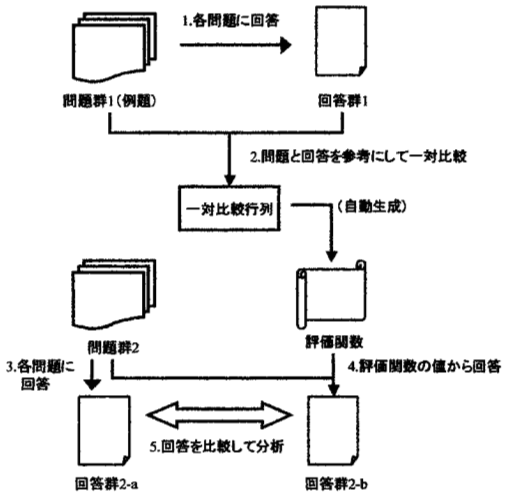


**図 5** 評価のプロセス

当する．問題毎に各ソースコードの外的振る舞いは等しく，処理速度とメモリ消費量は問題にならないものであり，保守性の観点のみで優劣を判断する．全ての問題で，利用するメトリクスの間にトレードオフが存在するので，評価関数によって判断の結果は変わる．

用意したメトリクスのセットとソースコード選択問題を用いて，提案手法の模擬的な実践とその結果の評価を行った．そのプロセスの全体像を図 5 に示す．

評価関数の作成は，例題を利用して行う．まず，一対比較を行う際に参考にするための例題のセットである問題群 1 の各問題に対して，それぞれ二つのうちどちらの保守性が優れていると考えるかを回答する．次に，得られた回答と，問題群 1 の各ソースコードとそのメトリクスの値を参考にして，一対比較行列を作成する．その際には，メトリクスのセットの一部のみを利用した場合のものも含め，重み付けの結果を得ることができる．また，得られた重みから評価関数を作成し，問題群 1 のソースコードに対する評価値の比較をすることができ，その結果から一対比較の内容を再検討することができる．

作成した評価関数についての評価を行うために，作業の順序に注意して以降のプロセスを行う．一対比較行列の作成を終えてから，問題群 2 の各問題に対して，問題群 1 と同様に回答し，回答群 2-a を得る．回答群 2-a を得てから，作成した評価関数を問題群 2 の各問題に適用して，その値に基づく回答を得る．ここで得られる回答群を 2-b とする．最後に回答群 2-a と回答群 2-b の内容を比較し，分析する．作業の順序から，評価関数と，問題群 2 の内容及びそれに対する作業者の判断は互いに影響しない．従って，回答群 2-a と回答群 2-b の内容が一致するかどうかによって，評価関数が開発者の選択の方針に沿っているかどうかを推定することができる．

今回の評価では，著者らが関数を作成した．問題は，問題群 1 として 15 問，問題群 2 として 10 問を用意した．

### 4.3 用意したメトリクス

評価項目として利用するために，以下の 8 種類のメトリクスをプロジェクト全体を対象にするものに加工し，評価項目とした．なお，一般的でないメトリクスの名称は，本稿において暫定的に与えた．

**LOCf** ファイルの行数から import 文の数を引いたものである．ファイルの行数が増えると，その内容を一度に視野に入れることが難しくなるため，保守性が悪化する．import 文は一般に，開発中に読むことが必要になる機会が少ないので，ファイルの行数自体よりも import 文の数を引いたものの方が，開発者にとっての実質的なファイルの行数に近いと推定し，このメトリクスを用意した．

**NOS** メソッドに含まれる文の数である．文の数が多いメソッドは処理の量が多く，理解しづらいという傾向がある．メソッドの抽出などによって理解しやすくなった場合，それを NOS の減少として検出できる．

**HOS** メソッド内の文一つを計測対象のモジュールとして，Halstead の volume[4] を計測したものであり，文の規模を示す．NOS を評価基準にしていると，多くの処理を一つの文にまとめることが望ましいと判断される可能性がある．しかし，それは逆に読解を困難にすることであり，このメトリクスによってその点を評価する．

**WMC** C&K メトリクスの一つで，クラスの複雑さを示すものである．機能を持ち過ぎているクラスから機能をほとんど持たないクラスへ機能を移すことによって，内容の理解が容易になったことなどが，WMC の減少として検出できる．

**CC** サイクロマティック複雑度[5] である．メソッド内の分岐の数に 1 を加えたものであり，メソッドの複雑さの指標となる．

**NOP** メソッドの引数の数である．引数が多いメソッドは，呼び出しの部分の可読性が悪くなるなどの問題がある．そのようなメソッドを無くしたことの効果を，このメトリクスによって検出する．

**RFC** C&K メトリクスの一つで，計測対象のクラスが持っているメソッドと，それらのメソッドから

呼び出される可能性のあるメソッドの数の合計である．他のクラスのメソッドを多く呼び出しているクラスは，処理内容がまとまっておらず理解や再利用が難しい傾向がある．

**CBO** C&K メトリクスの一つで，クラス間の結合度である．不要な結合を無くしたことの効果などを，このメトリクスによって検出する．

### 4.4 結　　果

#### 4.4.1 評価関数の作成

ソースコードの変更の選択の例題を利用するプロセスにより，評価関数を作成することができた．

例題に回答し，そのメトリクスの値を確認したところ，作成者の判断とうまく対応していないメトリクスがあった．作成者が，自分がクラスの大きさについて評価する際に，行数を重視し，WMC はあまり意識していないと感じた．また，RFC や CBO は，それらが示す品質に関して，作成者が優れていると判断した選択肢の方が値が悪い例があった．

まず，上記の 3 種類以外の 5 種類のメトリクスを対象にして一対比較を行い，評価関数を作成した．C.I. は 0.0035 であり，一対比較の整合性に問題は無いと判断した．この関数を例題に適用したところ，15 問のうち 14 問で，作成者による判断と一致する判断が得られることが確かめられた．

続いて，行列の修正を行った．まず，修正前の一対比較行列について，評価項目の重みから得られる，行列の各成分の理想的な値と，指定した値との誤差を調べ，一番大きかった部分を再検討した．また，前述の問題が CBO に対して軽微だと考えた RFC を一対比較の対象として追加した．ただし，RFC の一対比較の値がうまく浮かばなかったため，NOS と同程度にすると良さそうだと感じたことのみに基づき，RFC の行の値を全て NOS の行の値と同じにした．これらの二つの修正はいずれも，判断が一致しなかった例題の選択肢の評価値の差を縮めるものであったが，評価値の大小は変わらなかった．判断が一致していた 14 問についても評価値の大小は変わらなかった．なお，行列の成分の値を他の成分の値に基づいて決定した場合，C.I. の値が低くても，それを根拠に行列の信頼性が高いと判断することはできない．今回のプロセスでは，修正前の行列の C.I. の低さを信頼性の根拠とみなした．

無理に全ての問題で判断が一致する評価関数にすると，かえって重み付けの妥当性が失われ，未知の問題に対する判断が作成者による判断と一致する確率が下がるのではないかと推測したため，ここまでで評価関数の修正を終了した．作成した一対比較行列 $A$ と，そこから得られた重みを表すベクトル $\boldsymbol{w}$ を図 6 に示す．評価関数の作成にかかった時間は，1 時間 41 分であった（例題に対する作成者による回答にかかった時間は含まない）．

**表 1**　各問題に対する判断の理由

| 回答が一致した問題 | | 回答が一致しなかった問題 | |
|---|---|---|---|
| 理由 | 数 | 理由 | 数 |
| メソッドの規模 | 3 | ポリモーフィズム | 2 |
| クラスの規模 | 1 | による分岐 | |
| 委譲メソッドの有無 | 3 | メソッドの所属 | 1 |
| 合計 | 7 | 合計 | 3 |

#### 4.4.2 回答の比較

10 問の問題に対し，評価関数の作成者が回答した後にその関数による判断結果を取得し，各問題について回答を比較したところ，7 問が一致していた．問題を回答が一致したものと一致しなかったものに区分し，それぞれについて作成者が回答した際の判断の理由をまとめたものを表 1 に示す．

回答が一致したものとしなかったものとで，判断の理由の性質が異なっている．回答が一致した問題では，モジュールの規模やソースコードの分量に関する性質が良いと評価したものを選択している．回答が一致した 7 問のうち 4 問は，モジュールが小さいことを評価している．残りの 3 問は，委譲メソッドが無いことを評価しており，その理由の一つとして，委譲メソッドが無い方がソースコードの分量が少なくなることがある．一方，回答が一致しなかった問題では，モジュールの構成が良いと評価したものを選択している．回答が一致しなかった 3 問のうち 2 問は，選択した方のソースコードではポリモーフィズムによる分岐になっている部分が，選択しなかった方ではそうなっていない点が判断の根拠となった．残りの 1 問は，選択肢間でメソッドの移動が行われており，移動しているメソッドが所属しているクラスが妥当だと判断したものを選択した．なお，例題で回答が一致しなかったものも，モジュールの構成が良いと評価したものである．

### 4.5 評価と考察

判断の不一致の原因は，関数の作成者の判断の理由となった性質を，使用したメトリクスではうまく捕捉できないことであると考える．判断が一致しなかった問題は全て，モジュールの構成が良いと評価したものを選択している．用意したメトリクスの中で，モジュールの構成について計測できるものは，RFC と CBO である．しかし，これらのメトリクスの値は，計測の対象としている性質について，作成者が考える優劣に反

$$A = \begin{array}{c} \\ \text{LOCf} \\ \text{NOS} \\ \text{HOS} \\ \text{CC} \\ \text{NOP} \\ \text{RFC} \end{array} \begin{array}{c} \begin{array}{cccccc} \text{LOCf} & \text{NOS} & \text{HOS} & \text{CC} & \text{NOP} & \text{RFC} \end{array} \\ \begin{pmatrix} 1 & 1/4 & 5 & 1/20 & 1/12 & 1/4 \\ 4 & 1 & 20 & 1/5 & 1/3 & 1 \\ 1/5 & 1/20 & 1 & 1/100 & 1/60 & 1/20 \\ 20 & 5 & 100 & 1 & 2 & 5 \\ 12 & 3 & 60 & 1/2 & 1 & 3 \\ 4 & 1 & 20 & 1/5 & 1/3 & 1 \end{pmatrix} \end{array}, \boldsymbol{w} = \begin{array}{cc} & \\ \text{LOCf} & \\ \text{NOS} & \\ \text{HOS} & \\ \text{CC} & \\ \text{NOP} & \\ \text{RFC} & \end{array} \begin{pmatrix} 0.024 \\ 0.096 \\ 0.0048 \\ 0.50 \\ 0.28 \\ 0.096 \end{pmatrix}.$$

**図 6** 作成した一対比較行列と得られた重み

する測定結果となる例が存在し，CBO は一対比較の対象から除外された．そのため，評価関数の値に，選択したものの長所がうまく反映されず，判断が一致しなかったと考えている．一方，二つの判断方法で結果が一致した問題は，モジュールの複雑さやソースコードの分量に関する性質が良いと評価したものを選択している．選択に影響した性質について計測できるメトリクスとして，LOCf，NOS，HOS，CC があり，これらによって選択したものの長所が評価関数の値に反映されたと見ることができる．

また，判断結果が一致した問題は，モジュールの複雑さやソースコードの分量のみの間にもトレードオフを含んでいたが，それでも一致した結果を得られたことから，これらの性質に関しては，トレードオフがあっても，概ね作成者の価値観に合った判断結果を得られる評価関数ができたと推測した．

以上のことから，パラメータとするメトリクスで捕捉できる性質に関しては，それらの性質の間にトレードオフがある場合でも，開発プロジェクトの方針に沿った判断結果を得られる評価関数の作成が可能であると考えた．

## 5. 議　　論

提案手法で有用な評価関数を作成するためには，関数のパラメータにするメトリクスとして適切なものを用意することが不可欠である．本節では，メトリクスの用意に関する議論を行う．

### 5.1 加重平均に対するメトリクスの適合

提案手法の評価関数の値は各メトリクスの値の加重平均であり，各メトリクスはこのことに対して適合するものである必要がある．代替案 $A_i, A_j$ のメトリクスの値の集合をベクトルで表したものをそれぞれ $\mathbf{v_i}, \mathbf{v_j}$ とし，評価関数を $f$ とすると，以下の式が成立する．

$$f(\mathbf{v_i} - \mathbf{v_j}) = f(\mathbf{v_i}) - f(\mathbf{v_j}) \qquad (5)$$

従って，各メトリクスの値の差が一定であれば，評価関数による判断は一定になる．メトリクスの種類毎に，1 ポイントの改善の価値が一定であれば，上記の評価関数の性質に適合する．この条件を満たす方法は確立できていないため，試行錯誤によって，条件を満たしている状態に近いメトリクスを用意する必要がある．既存のメトリクスを加工したり組み合わせたりすることは，そのための手段となると考える．

### 5.2 モジュール単位のメトリクスの加工方法

提案手法において，モジュールを計測対象とするメトリクスをソフトウェア全体を対象とするものに加工する方法は，得られる判断に強く関与する要因である．

平均や合計のように，全てのモジュールの値が統合結果に影響する方法が適切である．一般に，メトリクスの値について調査する際には，閾値を超えているものや，最も値が悪いものを注視することが多い．その理由は，問題があるモジュールの特定を調査の目的としていることである．一方，提案手法においては，品質の変化を評価することを目的としており，どのモジュールの値が変化しても品質に影響が及ぶ可能性があるので，統合結果に影響を与えるモジュールを限定することは不適切である．

値が小さいことが望ましいメトリクスの加工方法の一種として，モジュール毎の値に 1 より大きい指数を付加し，合計することを考えた．本稿で行った評価ではこの方法を用いた．この方法では，値が大きくなるほど 1 ポイントの差の価値が増すため，全体の合計が下がらなくても，値が大きいものの改善がなされれば評価値も改善される．合計の代わりに平均を採用すると，空のモジュールの追加によって評価値が改善されるという問題が生じる．

## 6. 関 連 研 究

メトリクスを統合した保守性の評価指標として，MI (Maintainability Index)[6] がある．MI では，ファイ

ルの行数などのモジュールを対象としたメトリクスの値のソフトウェア全体での平均値を利用する．従ってMIの値は，ソフトウェアの規模によらず，絶対的な評価値として利用できるが，モジュールを過剰に分割するだけで向上するため，変更の評価には適さない．

Bansiyaらはメトリクスの統合による品質モデルであるQMOODを提案している[7]．QMOODでは異なる種類のメトリクスを同様に扱うために，比較対象のソースコードを利用してメトリクスの値の正規化を行う．そのため，各メトリクスの相対的な重みが，正規化に利用するソースコードによって変化するという問題がある．

また，進化的計算などの探索手法によってリファクタリングを探索し，適切と見なされたリファクタリングを提示する研究が盛んに行われており，その際の評価関数として，メトリクスを統合した計算式が利用されている[8)~10)]．Harmanらは，リファクタリングの探索の際に，複数のメトリクスを評価項目として，得られた選択肢の集合の中でパレート最適解となるものを，適用すべきリファクタリングの候補とすることを提案している[8)]．ただし，複数存在するパレート最適解の中からの選択の方法については論じていない．本稿の提案手法はこの分野に応用できると考える．

## 7. おわりに

本稿では，ソフトウェアメトリクスと，階層分析法の一対比較行列による重み付けの方法を用いて，ソースコードの変更の選択を行う手法を提案した．提案手法では，ソフトウェアメトリクスをパラメータとする評価関数の値によって変更の選択を自動的に行うことにより，作業者に依存しない判断結果を得ることができる．評価関数の作成を一対比較行列を利用して行うことにより，開発プロジェクトの方針を判断結果に反映させる．提案手法を評価するための事例を用意して調べたところ，ソースコード変更の選択の問題に対する，評価関数による回答と関数の作成者の回答が，10問中7問で一致した．回答が一致しなかった3問はいずれも，作成者の判断の根拠が評価項目として用意したメトリクスの値に表れるものではなかった．

今後の課題として，追加実験と扱う品質の範囲の拡大を考えている．

現時点では，著者らが行った評価によって提案手法の有用性を確認しているが，被験者実験を重ね，より信頼性の高い評価をする必要がある．また，大規模開発に対する適用事例を得る必要がある．提案手法による解決を意図している問題は規模が大きいソフトウェアほど重要になるので，大規模開発において手法が有用であることが求められる．

提案手法は現時点では保守性のみを扱っているが，より多くの品質を対象にできることが望ましい．そのための方法として，プロファイラから得られる情報を評価関数のパラメータに加えることで，実行時の効率を扱うことを考えている．保守性と実行時の効率のトレードオフが原因で変更の選択が難しくなる状況は多いので，手法の有用性が大きく向上すると考えている．

## 参考文献


1) Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).
2) Saaty, T.L.: *The Analytic Hierarchy Process: Planning Setting Priorities, Resource Allocation*, Mcgraw-Hill (1980).
3) Chidamber, S.R. and Kemerer, C.F.: A Metrics Suite for Object-Oriented Design, *IEEE Transactions on Software Engineering*, Vol.20, No.6, pp.476–493 (1994).
4) Halstead, M.H.: *Elements of Software Science*, Elsevier (1977).
5) McCabe, T.J.: A Complexity Measure, *IEEE Transactions on Software Engineering*, Vol.2, No.4, pp.308–320 (1976).
6) Coleman, D., Ash, D., Lowther, B. and Oman, P.: Using Metrics to Evaluate Software System Maintainability, *Computer*, Vol.27, No.8, pp. 44–49 (1994).
7) Bansiya, J. and Davis, C.G.: A Hierarchical Model for Object-Oriented Design Quality Assessment, *IEEE Transactions on Software Engineering*, Vol.28, No.1, pp.4–17 (2002).
8) Harman, M. and Tratt, L.: Pareto Optimal Search Based Refactoring at the Design Level, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp.1106–1113 (2007).
9) O'Keeffe, M. and O'Cinneide, M.: Search-Based Software Maintenance, *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pp.249–260 (2006).
10) Seng, O., Stammel, J. and Burkhart, D.: Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems, *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pp.1909–1916 (2006).