ソフトウェア工学 138−13
（２００２．７．１２）

機能多重度を用いた保守性見積もり手法の提案と評価

今井 健男 †　片岡 欣夫 †　深谷 哲司 †
† 株式会社 東芝 研究開発センター システム技術ラボラトリー
E-mail: {tateo.imai, yoshio.kataoka, tetsuji.fukaya}@toshiba.co.jp

概 要

コードクローン (コピー＆ペースト等によってソースコード中に生じる類似したコード片) はしばしばソフトウェアの修正・拡張といった保守作業時に問題を生じる。これに対し、我々はコードクローンを含有すると思しき関数の散乱の度合いを示す機能多重度 (Functional Redundancy、FR) を提案する。これにより、コードクローンの観点から、対象となるソフトウェアが抱える保守作業の非効率性を見積もる事ができる。本稿では上記指標の提案と共に、ソフトウェアの保守性改善を目的としたリファクタリング作業の前後で提案指標の変化を調べ、提案指標の有用性について議論する。

Estimating Software Maintenance Cost Using Functional Redundancy Metric

Takeo Imai † Yoshio Kataoka † Tetsuji Fukaya †

†System Engineering Laboratory
Corporate Research and Development Center
Toshiba Corporation

Abstract

Source code copying for reuse (code cloning) causes difficulty when software is modified or enhanced. Here we aim to estimate the maintenance cost caused by clones. We propose a novel metric, Functional Redundancy (FR): A degree of propagation of clone-potential functions. This metric estimates influence of cloned codes over the maintenance cost. In this paper, we describe the details of our proposal. We also apply our metric to a empirical software before and after its refactoring, and discuss effectivity of our metric.

1　はじめに

コードクローンとは、同じソフトウェア中に存在する、類似した、あるいは全く同一のソースコード片のことである。これは以下のような理由から、殆どのソフトウェアにおいて散見される。

1. 開発者が、既に実装した機能と同様の機能を実装する際に、先に実装してあるソースコードの内容を複製する事で実装を済ませ、実装作業の効率化を図る。
2. ある処理を行う為のコードの実装方法が定型化・パターン化している為、開発者がそのパターンに従って記述する。

一方で、このようなコードクローン (以降、単にクローンとも呼ぶ) が同一ソフトウェア中に散在する事は、ソフトウェア開発プロジェクトに悪影響を及ぼす事が良く知られている。ここでは文献 [3] から、そのような悪影響、特にソフトウェアの保守作業を悪化させる現象に関する記述を引用する。

- The same software bug reoccurs throughout software despite many local fixes.
- Code reviews and inspections are needlessly extended.
- It becomes difficult to locate and fix all instances of a particular mistake.
- Software defects are replicated through the system.
- Developers create multiple unique fixes for bugs with no method of resolving the variations into a standard fix.

ここで挙げた項目は、それぞれソフトウェアの修正や拡張といった、ソースコードの改変作業を非効率化させるも

のだと言える。開発者がソースコードを改変しようとする際、該当する改変個所のクローンに相当するコード片は、該当個所と同一の機能を果たしている以上、やはり同様の改変を施さなければならない可能性を孕んでいる。例えば、ある不具合を修正する目的であるコード片を改変し、そのコード片と同様の機能を果たしているクローンを改変しなかった場合、ここで修正したはずの不具合が後々にまた発生する事になる。

このようなコードクローンに対して、従来から多くのコードクローンに関する研究が行われてきたが、その殆どはクローンの検出技術に関するものであった [1][2][5][6][7]。一方で、上述の検出技術を用いるか否かに関わらず、コードクローンがソフトウェアの保守作業にどれだけ悪影響を及ぼすかを見積もる研究に関しては、我々の知る限り殆どされてこなかった。

我々は、あるソフトウェアに対して、その保守作業にコードクローンが与える影響の度合いを定量的に見積もる指標として**機能多重度**(Functional Redundancy, FR) を提案する。これは、コードクローンを含有すると思しき関数の散乱の度合いを示すものである。

2 機能多重度の概要

以下では、機能多重度の定義について、背景にある考えも含めて詳説する。

2.1 コードクローンと保守性に関する考察

まず、コードクローンが保守作業に関して、どのような影響を与えるかを、ソフトウェアの改変について簡易なプロセスモデルを想定した上で、定性的に考察してみる。図 1 は、本モデルの処理の流れを示したものである。

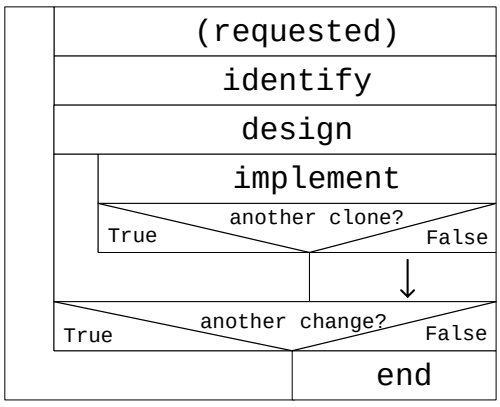

図 1: クローンを考慮したソフトウェアの改変

クローンを考慮しない単純なモデルでは、ある 1 つの変更要求 (request) に対して

1. 改変個所の特定 (**identify**)
2. 改変コードの設計 (**design**)
3. 改変コードの実装 (**implement**)

の各フェーズが手順として順に、1 度ずつ発生する、と考えられる。

ここでクローンを考慮すると、まず **identify** フェーズにおいて、本来改変したい個所に加え、そのクローンの個所も予め、目視や上述のクローン検出技術などを用いて特定しておくべきである。

そして次の **design** フェーズにおいて、クローンも併せた上で改変の内容が検討される。この検討内容は、例えばクローンの削減や共通化の検討などがなされるかもしれないし、共通化しない場合おいても、各クローンに対して同様の改変を加えるか否か (いくつのクローンに改変を加えるか) の判断が必要になる。

最後に、**implement** フェーズにおいて、検討した改変を実際にソースコード上に展開する。ここで注意すべきは、クローンを考慮しないモデルにおいてこのフェーズでの作業が 1 回だったのに対し、クローンを考慮する場合は、この作業が複数回、**design** フェーズで決定した、改変すべきクローンの数だけ発生する、と言う事である。

従って、クローンを考慮した改変は考慮しない場合に比べて全体の作業工数が増加すると考えられるが、特に、我々の考慮したモデルにおいては **implement** の作業が最も増大する。具体的に言えば、上述の通り「改変すべきクローンの数」(すなわち、最大で全てのクローン数) だけ繰り返し作業 (図 1 における、内側のイタレーション) が発生する事になる。

さらに別の変更要求があった場合には、その要求に応じて、やはり同様の作業が発生するが、ここでクローンの観点から、次の 2 点に注意しなければならない。

1. **identify** フェーズにおいて、往々にしてクローンの検出に失敗する場合がある。この場合、検出漏れしたクローンが後々に不具合などを起こし、もう一度変更要求を発生させる可能性がある。
2. 検出漏れがないとすれば、変更要求が発生する数は、多くとも発見された**クローンのグループ数**だけ発生する可能性がある。

そしてこの場合、作業は新たな変更要求を通して **identify** フェーズから **implement** フェーズまでの全ての作業が繰り返し発生してしまう。(図 1 における、外側のイタレーション)

従って、このモデルを元にすれば、コードクローンはそのグループ数と、各グループ内に含まれるクローン数が保守作業の発生回数に大きく影響していると考えられる。

以上から、我々はクローンのグループ数と、各グループ内に含まれるクローン数 (これらを合わせて**クローンの散乱の度合い**と呼ぶ) を、保守作業の非効率性を見積もる基本概念とする。

2.2 機能多重度の基本概念

2.2.1 関数と類似度

ここで、我々はクローンの散乱の度合いを見積もる単位として、手続き型言語における関数 (手続き) を 1 つの単位とみなした。それは以下のような、経験的な理由による。

- ソースコードの改変は、ソフトウェアの機能をベースにしてなされる事が多い。そして関数はソースコード内で、最も明示的に機能を体現する構成要素の 1 つである。
- ソースコードの複製によるコードクローンの作成は関数規模、あるいはそれより小さな規模でなされる事が多い。また、関数をまたがるようなコード片の複製はあまり行われない。

しかし、当然ながら、関数の一部のみがクローンとなっている場合がある。また、第 1 節冒頭にあるクローン発生の理由から、クローン同士が同一のコード片になる場合も少ない。クローンとなるコード片は多くの場合、用途に合わせて識別子名が変更されたり、ステートメントが追加、もしくは削除されたりするケースが往々にしてあるからである。従って、クローンとなるコード片そのものをこの枠組みで扱う事はできない。

この問題を解決する為、我々は 2 関数間の**類似度**を導入する。類似度の定義は以下のとおりである。

> 関数 a, b の類似度 $S(a, b)$ とは、a, b の全行数のうち、共通となる構成要素の割合である。ただし、$S(a, b) = S(b, a)$ とする (すなわち、対称律が成り立つものとする)。

ここで構成要素とは、関数より小さい粒度となるプログラムの要素 (文字、字句、行など) である。
以降では便宜上、類似度を以下のように定義しておく。

$$S(a,b) = max\left(1 - \frac{common(a,b)}{LOC(a)}, 1 - \frac{common(a,b)}{LOC(b)}\right)$$

ここで、$LOC(f)$ は関数 f の全実行行数、$common(f, g)$ は関数 f, g に共通な行の数とする。

2.2.2 クローンの定義と機能多重度

そして、任意の 2 関数 f_1, f_2、類似度 s から、下記のように、間接的にクローンを定義する。

> $S(f_1, f_2) = s$ であるとき、f_1 **は確率** s **の可能性で** f_2 **のクローンを含有**している。

S は対称律を満たすので、上記の定義で f_1 と f_2 が入れ替わっても同様に成り立つ。

そして、前節で述べた基本概念を、上述のクローンの定義を用いて、**クローンを含有する関数の散乱の度合い**と置き換える。そして、この度合いを定量的に見積もる指標として**機能多重度**を定義する。

以降では、機能多重度の計算方法について述べる。

2.3 機能多重度の計算方法

2.3.1 関数と類似度の関係

関数と類似度の関係は図 2 のようなグラフで表記する事ができる。1 から 8 までの小円で示されたノードは関数を、小円の間に引かれたエッジは関数間の類似関係、エッジ上の [0..1] 間の数値が類似度を示している。

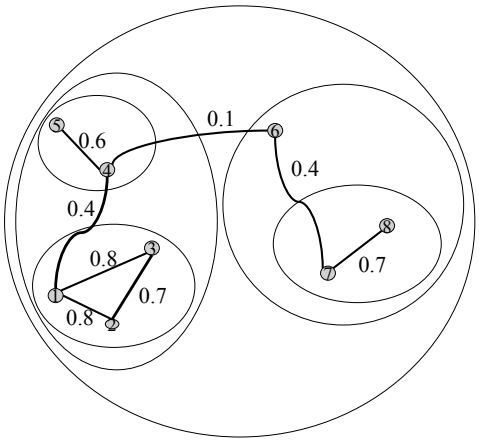

図 2: 関数と類似度で構成されるグラフ

類似度から、関数は図 2 中の楕円で示されるような、階層的な集合を形成する。この階層は、例えば下記のような手順をもって構成する事ができる。

1. まず、閾値を 0 と置く。
2. 閾値を上げる。
3. ある類似度の値が閾値を下回った時点で、その類似度に関連するエッジを切断する。
4. もし、3 において、ある部分グラフが他と連結しなくなったとき、その時点で連結しているノード (関数) で集合を形成する。
5. 閾値が 1 になるまで 2 以降を繰り返す。1 になった時点で終了する。

図 2 の例で言えば、閾値が 0.1 の時点で {1,2,3,4,5} と {6,7,8} の 2 集合が形成される。閾値が 0.4 の時点で {1,2,3}、{4, 5}、{6} および {7,8} の 4 集合が形成される。

2.4 FR ツリー

FR ツリーは、関数とその類似度によって構成される n 分木である。図 2 の例を FR ツリーで示すと、図 3 のようになる。

FR ツリーは、一般的な n 分木と同様、ノードとその親子関係によって構成され、**頂点**(親ノードを持たないノード)、**葉**(子ノードを持たないノード) 及び**節**(葉でないノード) を持つ。

このツリーにおける葉は、ソースコード中の関数 1 つにそれぞれ対応する。全ての関数が漏れなく葉に対応する。また、節は、葉または他の節を子として持ち、類似度を属性として持つ。

各節の属性となる類似度の値は、次のようにして決定される。

節 K が n 個の子を持っているとき、各々の子に対して

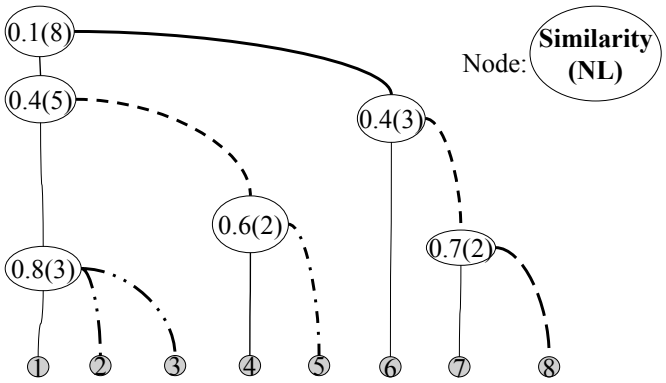

図 3: FR ツリー

- 子が葉であれば、その葉自身
- 子が節であれば、その節が持つ葉の中の 1 つ

を選び出し、選び出された n 個の葉から成る集合 $leaf(K) = \{k_1, k_2..., k_n\}$ を考える。

さらに、任意の類似度 s が与えられたとき、任意の $k \in leaf(K)$ に対して、

$$k \text{ との類似度が } s \text{ である } k_i \in leaf(K) \quad (1 \leq i \leq n, k \neq k_i)$$

が必ず存在するような $leaf(K)$ を $conleaf(K, s)$ と定義する。

節 K が属性として類似度 s を持つとは、以下の条件 1、2 が共に成り立つ事と同義である。

1. $conleaf(K, s)$ **が必ず 1 つ以上存在する。**
2. **任意の** $s_1 > s$ **に対して、**$conleaf(K, s_1)$ **は存在しない。**

さらに、FR ツリーは以下の規則に基づいて構築されるものとする。

> 親子関係を成す任意の 2 つの節 P と C について、各節が持つ類似度 $s(P)$ と $s(C)$ の間には
>
> $$s(C) > (P)$$
>
> という関係が常に成り立つ。

2.5 機能多重度の計算プロセス

2.5.1 FR ツリーの構築

図 4 は、FR ツリーの構築アルゴリズムを擬似コードで記述したものである。

入力として、全ての 2 関数の組とその間の類似度が与えられる。そして、各組が持つ関数を類似度に基づいて降順にツリーに組み込んでいく。

```
procedure MakeTree(pairs: array of Pair)
var
  T: Tree;
  N: Node;

begin
  sort_by_similarity(pairs, decreasing);
  for i := 1 to sizeof(pairs) do begin
    if T contains pairs[i].node[1] and
       T contains pairs[i].node[2] then return;

    for j := 1 to 2 do begin
      new(N);
      register(N, T);
      if T contains pairs[i].node[j] then
        add tree_top(pairs[i].node[j]) to N.child;
      else
        add pairs[i].node[j] to N.child;
    end
  end

  return;
end
```

図 4: Algorithm of Making FR Tree

2.5.2 機能多重度の計算手順

ここで、構築された FR ツリーに基づいて機能多重度を計算する。

ここで、機能多重度とは以下のような性質を満たしているべきである。

- 関数間の類似度が高い程、これらの関数が同一の機能を実装していると推測できる。すなわち、それだけ機能多重度が高くなる。
- 同じ機能を実装している (と推測できる) 関数が多いほど、機能多重度が高い。

従って、

1. 接点数が一定の場合、各節が持つ**類似度**が高いほど単調に増加する
2. 接点数、類似度が一定の場合、**同じ機能を実装していると推測できる関数の多さ**に合わせて単調に増加する
3. 対象となるソースコード毎に正規化できる (つまり、他のソースコードで計った値と比較可能である)

値を機能多重度の必要条件とする。

以上のような性質を反映させるように、FR ツリー中の各節が属性として持つ**類似度**と**節が持つ子の数**から、機能多重度を以下のように定義する。

定義 あるソースコード P に対して、機能多重度 $fr(P)$ を、次の式で定義する。

$$fr(P) \stackrel{def}{=} \sum_{N \in nodes_{int}(P)} (|child(N)| - 1) \times s(N) + 1$$

ここで

- $nodes_{int}(P)$ は P から作られる FR ツリーが持つ全ての節から成る集合
- $child(N)$ は N が持つ全てのノードからなる集合

である。

ここで注目すべきは、$fr(P)$ は、P に含まれる関数数によって正規化が可能な事である。

上記 $fr(P)$ の定義式において、右辺にある積項 (すなわち、$(|child(N)|-1)\times s(N)$ および '1') の総数は P に含まれる関数数に等しい。

よって、類似度の値域が [0..1] として, $fr(P)$ と P が持つ関数数 ($|func(P)|$) は次式のような関係を持つ。

$$fr(P) \leq |func(P)|$$

(等号は、全ての類似度が 1 に等しいとき、かつそのときのみ成り立つ)

以上から、**正規化した機能多重度**$fr\_norm(P)$ を次式で定義する。

$$fr\_norm \stackrel{def}{=} \frac{fr(P)}{|func(P)|}$$

例として、図 3 の場合で機能多重度 $fr(P)$ と、正規化した機能多重度 $fr\_norm(P)$ は以下のように計算される。

$$\begin{aligned} & fr(P) \\ = \;& 0.1\times 1+0.4\times 1+0.4\times 1+0.6\times 1+0.7\times 1 \\ & +0.8\times 2+1 \\ = \;& 4.8 \\ & fr\_norm(P) = 4.8/8 = 0.6 \end{aligned}$$

3 適用実験と評価

以上で述べた機能多重度の有用性を確認する為、我々は適用実験を行った。以降はその報告である。

3.1 実験の目的

本実験では、機能多重度によってソフトウェアの保守性を定量化できること、具体的には

1. 実際のソフトウェアで機能多重度が計測可能であること
2. 機能多重度が、ソフトウェアの保守性変化を反映すること

の 2 点を確認する。

3.2 実験の手順と対象

本実験は以下に示すような流れで行った。

1. あるソフトウェアに対して、まず機能多重度を計測する。
2. 対象ソフトウェアに対し、コードクローンの除去を中心としたリファクタリング[1]を施す。具体的には、文献 [11] にある
   - メソッドのパラメータ化
   - メソッドの抽出
   - メソッドのインライン化

   等の作業[2]を行い、コードの重複を排除する。
3. リファクタリング後のソフトウェアに対し、再度機能多重度を計測する。
4. リファクタリング前後での機能多重度を比較する。

対象は約 8K ELOC(Executable Lines Of Code)、2 モジュールから構成される C 言語で記述されたソフトウェアである。(表 1 参照)

	ELOC	関数数	ELOC/関数
モジュール A	3780	277	15.7
モジュール B	3887	247	13.6

表 1: 対象ソースコード (リファクタリング作業前)

このソフトウェアのうち

1. モジュール A 全体
2. モジュール B 全体
3. モジュール A のうち、リファクタリングによって変更された一部
4. モジュール B のうち、リファクタリングによって変更された一部

のそれぞれに対して機能多重度を測定した。なお、3 及び 4 は、実際の変更部分のうち、変更が明確に確認できたもののみを抽出した。

また、上記の手順 2 におけるリファクタリングがソースコードに与えた影響を判定する為、下記の指標も提案指標に併せて測定した。

- 総行数 (ELOC)
- 関数の総数
- 1 関数あたりの平均行数 (ELOC)
- McCabe サイクロマティック複雑度 [10] が 10 以上となる関数の数

提案指標は比較の為、正規化した機能多重度で示す。以降は簡単の為、正規化した機能多重度を単に「機能多重度」と呼ぶ。

[1] リファクタリングとは、文献 [11] によれば「ソフトウェアの外部的振る舞いを保ったままで、内部の構造を改善していく作業」を指す。

[2] ただし、「メソッド」は C 言語の関数に置き換える。

	総 ELOC	関数数	ELOC/関数	機能多重度 (%)	McCabe ≥ 10
リファクタリング前	3780	277	13.6	48.5	12
リファクタリング後	3530	237	14.9	46.8	15
比較	-250	-40	+1.3	-1.7	+3

表 2: 測定結果 (モジュール A:全体)

	総 ELOC	関数数	ELOC/関数	機能多重度 (%)	McCabe ≥ 10
リファクタリング前	3887	247	15.7	60.8	9
リファクタリング後	3734	220	17.0	59.7	17
比較	-153	-27	+1.3	-1.1	+8

表 3: 測定結果 (モジュール B:全体)

	総 ELOC	関数数	ELOC/関数	機能多重度 (%)	McCabe ≥ 10
リファクタリング前	748	53	14.1	62.4	2
リファクタリング後	663	31	21.4	52.0	4
比較	-85	-22	+7.3	-10.4	+2

表 4: 測定結果 (モジュール A：変更部分)

	総 ELOC	関数数	ELOC/関数	機能多重度 (%)	McCabe ≥ 10
リファクタリング前	784	38	20.6	75.6	0
リファクタリング後	398	22	16.1	35.0	2
比較	-386	-16	-2.5	-40.6	+2

表 5: 測定結果 (モジュール B:変更部分)

3.3 実験結果

以上のような実験で得られた測定結果を、表 2 から表 5 に示す。

3.3.1 リファクタリング効果の確認

まず、リファクタリングの前後で、ソフトウェアがどのように変わったかを評価する。

モジュール A、B のいずれにおいても、ソースコード全体の ELOC、関数数は減少している。

実際の作業内容を我々が目視で確認すると、単純な「メソッド (関数) のパラメータ化」による関数の統合が最も多くなされていた。これにより複数の関数が 1 つにまとめられたので、ELOC、関数数が減ったものと推測される。

モジュール A、B ともに関数あたりの平均 ELOC 増大、一部関数の複雑度増大が見られるのも、「メソッドのパラメータ化」作業が最も多く行われた事の傍証になるであろう。つまり、複数の関数が統合され、その際にパラメータに依存するコードや条件分岐が統合後の関数に追加された事が、測定値に反映されていると思われる。

「メソッドのパラメータ化」はコードの重複を排除する効果がある [11]。よって、コードクローンがリファクタリングの前後で除去されている事が推測され、今回のリファクタリングが目的に適ったものであったと判断できる。

3.3.2 機能多重度指標の評価

これに対して、機能多重度の測定結果はリファクタリング前後でいずれも減少していた。モジュール全体では A、B 共にパーセンテージ表記で 1〜2 程度の僅かな減少であるが、変更した部分のみをみれば、モジュール A では機能多重度が 10.4、モジュール B では 40.6 の大きな減少が見られた。これは、前節から得られた、リファクタリング前後でのコードクローンの減少という結果に合致している。

3.4 考察

本適用実験の結果から、機能多重度が実際のソフトウェアで計測できる事、およびコードクローンの減少を正しく反映している事が確認できた。コードクローンを中心に考えればリファクタリング前後で保守性は向上していると考えられるから、機能多重度は保守性の評価指標として有効であったといえる。

ただし、リファクタリング前後でサイクロマティック複雑度が上昇している点を考慮すれば、機能多重度のみでは保守性を評価できない事も暗に示している。3.3.1 節での考察から、コードクローン除去の作業は複雑度の上昇を起こしやすい、とも考えられる。もしそうだとすれば、機能多重度とサイクロマティック複雑度は時にトレードオフとなる可能性もある。

ここで一つ、留意しておかなければならない点は、ELOC・関数数と機能多重度との相関性についてである。

本実験では ELOC、関数数、機能多重度の全てがリファクタリング前後で減少しているが、これは上述のように「メソッドのパラメータ化」作業が他の作業に比して多かったためであり、必ずしも、ELOC・関数数による変化が機能多重度の評価やクローンの数の変化と一致する訳ではない。一つの反例として、以下のようなリファクタリング作業を考える。

コード片 a, b, c から成る関数 $f_1\{a, b, c\}$、d, b, e から成る $f_2\{d, b, e\}$ があったとする。a から e の任意の 2 つは互いに類似度 0 とする。つまり、f_1 と f_2 はクローン b を互いに含有している。

f_1 と f_2 の類似度が仮に 70%だったとすると、機能多重度は 85(パーセンテージ表記) となる。

これに対して、クローン b を除去するために「メソッドの抽出」作業によって、f_1、f_2 のそれぞれから b だけ括りだして関数化し、「メソッドのパラメータ化」で共通化すると、

$$f_1'\{a, c\}, f_2'\{d, e\}, f_3\{b\}$$

の 3 つの関数が得られる。

f_1'、f_2'、f_3 は、単純に考えればどの 2 つも類似度 0%であり、これから機能多重度は 33 となる[3]。つまり機能多重度は減少するが、関数数は 1 増加している。また、ELOC は単純に考えれば減少するが、実際には f_1'、f_2' 中から f_3 を呼び出すコード等が追加されるので、場合によっては ELOC が増加する事もある。

4 関連研究

先に述べたように、コードクローンの検出技術はこれまでに多く提案されている [1][2][5][6][7] が、我々の研究はこれらと競合するものではない。2.1 節で述べたようなプロセスモデルを考慮すれば、彼らの提案する技術やその応用と、我々の機能多重度は相互に補完しあうものである。

1 つ、我々の研究について着目すべきは、クローンの検出を厳密に行わずに、クローンの影響を定量的に評価している点である。クローンの検出には一般に大きな計算量を伴う。その様な計算を伴わなくても良い点が提案手法の長所の 1 つである。

Mayrand らの研究 [8][9] は我々と同じく、コードクローンに着目したソフトウェアの品質評価を目的としている。彼らのアプローチの特徴は、コードクローンをその修正パターン毎に階層化している点にある。Balazinska ら [1] も Mayrand らのアプローチを踏襲しており、クローンを 18 パターンに分け、それをソフトウェア改善時の指針に使用している。

彼らのアプローチに比しての提案手法の長所は、品質評価のスコープである。我々の手法ではソフトウェア全体の品質を 1 つの指標値で示せるのに対し、彼らのアプローチでは 2 関数間の関係のみに留まっている。評価結果の理解性、一覧性という観点からすれば、我々の提案手法が勝っていると考える。

井上らの文献 [12] では、クローンの存在を利用した保守性の指標として、nonclone-SLOC(プログラムテキストからコードクローンを取り除いた行数) の可能性を示唆している。我々は、本稿とは別の研究にて、これと同様の指標[4]と提案指標との比較を行っている [4]。そこでは、この 2 つの指標は強い相関を持つ一方、提案指標はクローンのサイズよりも、複製された数を強く反映する傾向がある事を示している。

[3]正規化する前に必ず 1 が加算される為、正規化後も 0 にはならない。

[4]実際には、全プログラムテキスト中のコードクローンの割合

5 まとめと今後

我々は、コードクローンの観点から、ソフトウェアの保守作業にかかる非効率性を見積もる為の指標である**機能多重度**(Functional Redundancy, FR) を提案した。この指標は、2 関数間の類似度を用いて関数をクラスタ化し、得られたクラスタ (FR ツリー) の構成を定量的に測定する事で、コードクローンを含有すると思しき関数の散乱の度合いを評価するものである。

本稿では上記指標の提案と共に、この指標の有用性を評価する為、実際に行われたソフトウェアの保守性改善プロジェクトを対象として適用実験を行った。具体的には、コードクローンを除去する目的でリファクタリングされた 8K-ELOC(Executable Lines Of Code) 規模のソースコードに対して、リファクタリング前後のソースコードを提案指標で評価し、指標値に変化が見られるかどうかを試した。結果、リファクタリング以前のコードに比べ、リファクタリング後の指標値が低下した。

この結果から、保守性改善作業によって得られたソフトウェアの保守性向上と提案指標の変化の間に相関があると見られ、ソフトウェアの保守性を定量的に見積もる 1 手法として提案指標の評価が有用である事が示された。

今後は提案指標に対して更なる評価を試み、より確かな検証を進める。最終的には、指標に対する一般的指針 (例えば、どの程度の値が示せれば保守性の高低が言えるのか、あるいはどの程度減少すれば保守性が改善されたと言えるか、など) の提示も目標としたい。

また、今後は提案する機能多重度を単なる保守性評価に用いるだけでなく、実際の改善作業の効率化に用いて行く事も思慮に入れている。これについては、我々が先に発表した論文 [4] に手法の提案のみ行っている。

参考文献

[1] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *Metrics'99*, 1999.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, and Marcelo Sant'Anna Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the 1997 International Conference on Software Maintenance (ICSM'97)*. IEEE Computer Society Press, 1997.

[3] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormickIII, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley Computer Publishing, 1998.

[4] Takeo Imai, Yoshio Kataoka, and Tetsuji Fukaya. Evaluating software maintenance cost using functional redundancy metrics. In *the proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC 2002)*. IEEE Computer Society Press, 2002. (to be appeared).

[5] J Howard Johnson. Identifying redundancy in source code using fingerprints. In *CASCON'93*, 1993.

[6] Toshihiro Kamiya, Fumiaki Ohata, Kazuhiro Kondou, Shinji Kusumoto, and Katuro Inoue. Maintenance support tools for java programs: Ccfinder and jaat. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001)*. ACM Press, 2001.

[7] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of The Eighth Working Conference On Reverse Engineering 2001*, 2001.

[8] Jean Mayrand, Bruno Laguë, and John Hudepohl. Evaluating the benefits of clone detection in the software maintenance activities in large scale systems. In *WESS '96*, 1996.

[9] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Management 1996*. IEEE Computer Society Press, 1996.

[10] Thomas J. McCabe. A complexitjy measure. *IEEE Transactions on Software Engineering*, Vol. 2, pp. 308–320, 1976.

[11] マーチンファウラー. リファクタリング—プログラムの体質改善テクニック. ピアソン・エデュケーション, 2000.

[12] 井上克郎, 神谷年洋, 楠本真二. コードクローン検出法. コンピュータソフトウェア, pp. 47–54, Sep. 2001. 日本ソフトウェア科学会 編.