4R−6

リファクタリングによるソフトウェア品質改善と品質劣化の予防 (3) リファクタリング実施効果の定量化手法

片岡 欣夫,　本間 昭次,　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
{yoshio.kataoka|akitsugu.homma|tetsuji.fukaya}@toshiba.co.jp

1　はじめに

リファクタリングは「プログラムの意味・実現機能を変更すること無しに, プログラムの構造を変更する技術」と定義され, プログラムの保守性向上を目的とする技術である. リファクタリングを行うことにより,

- 常により良いシステム設計を保てる
- 可読性, 保守性の高さを保てる
- 潜在的バグを発見できる
- 拡張・改良の工数を短縮できる

など, 保守性や拡張性を向上させる改善効果や, 予め拡張性を確保する予防効果が期待される. また文献[1]等に紹介されているように, リファクタリングは今やソフトウェア開発技術の基礎技術であり, ソフトウェア開発者に求められる必須技術となりつつある.

これらの有効性が一般的に認知されているにもかかわらず, 稼働中のプログラムに対して変更を加えることで新たな誤りを混入するかも知れないと言う危惧などから, リファクタリングが現場で積極的に行われることは少ない.

我々は, リファクタリングの適用効果を定量化することによりその有用性を示し, コスト対効果の議論を行いやすくすることで, リファクタリングの現場での積極的活用を目指している. 本稿ではこのリファクタリング適用効果の定量化手法について述べる.

2　リファクタリングの効果

リファクタリングの効果として保守性向上が上げられるが, これをどのように定量化するかがポイントになる. ここでは「保守性の低さ」というものを定量化することを考える. 保守性の低さを定量化することが出来れば, その数値の改善度合をもってリファクタリングの効果とすることが出来る. ここで考える「保守性の低さ」は, プログラム全体としての評価を行う必要は無い. 一般にリファクタリングはローカルなプログラム部分に対しての変更作業となるため, 対象となるプログラム部分の保守性の低さを考慮することになる. 即ち, プログラム中の保守性の低い部分が, リファクタリングによってどの程度保守性が向上するかと言うことを示すことになる. 以下では保守性の低さを, 変更作業に要する工数の多さと定義する.

以下では対象言語をJavaとし, プログラム部分の単位をメソッドとして, 保守性の議論を進める. 即ち, Javaプログラムのメソッド単位でのリファクタリング効果の定量化方法について議論する.

3　メソッドの結合度

メソッド間の結合は,

- メソッド A がメソッド B へ p 個のパラメタを渡している.
- メソッド A がメソッド B が参照している c 個のクラス変数に書き込んでいる.
- メソッド B がメソッド A の返り値を使っている.

などに大別される. メソッド A が多くのメソッドと結合関係を持つ程, メソッド A の修正時に考慮すべき依存関係が多く変更作業に要する工数が多くなる, 即ち保守性が低いと言える. このことからメソッド A に関するリファクタリングについては, メソッド A の他のメソッドとの結合度を少なくする度合をもって, 保守性向上の効果とすることが出来る. また結合度を考慮する場合, クラス間結合は, クラス内結合よりも保守コストがかかる. 具体的には3, 3のケースでは, クラス内結合よりクラス間結合の方が重くなるように係数を考慮する必要がある. これらを踏まえて, メソッド A とメソッド B の**結合度**を以下のように定義する.

クラス内結合度 $p+c+1$

クラス間結合度 $p+C_{cv}c+C_{rv}$ $(C_{cv}>C_{rv}>1)$

以下ではこの定義に基づき, 具体例を用いてリファクタリング効果の定量化を試みる.

4　実験

4.1　定量化の具体例

文献[2]より, "Move Method" というリファクタリングを例に取る. "Move Method" は, 例えばあるメソッドが定義されているクラス内よりも, 他のクラスからより頻繁に使用されているメソッドを移動すると言うリファクタリングである.

"Software Quality Improvement and Deterioration Prevention using Refactoring – Quantification of Refactoring Effect –" by Josh KATAOKA, Akitsugu HOMMA, and Tetsuji FUKAYA, System Engineering Laboratory, Corporate Research and Development Center, TOSHIBA Corporation

```
class Account...
  double overdraftCharge() {
    if (_type.isPremium()) {
      double result = 10;
        if (_daysOverdrawn > 7)
          result += (_daysOverdrawn - 7) * 0.85;
        return result;
    }
  }

  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result += overdraftCharge();
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
```

この例では, AccountType の種別 (プログラム中で isPremium() 等で判別される) が増えることを想定し, overdraftCharge() を AccountType へと移動し, 以下のような実装に変更している.

```
class AccountType...
  double overdraftCharge(int daysOverdrawn) {
    if (isPremium()) {
      double result = 10;
      if (daysOverdrawn > 7)
        result += (daysOverdrawn - 7) * 0.85;
      return result;
    }
    else return daysOverdrawn * 1.75;
  }

class Account...
  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result +=
        _type.overdraftCharge(_daysOverdrawn);
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
```

AccountType の種類を n 個として, 関連する範囲内での結合度の総和を比較してみる. overdraftCharge() を中心に結合を調べてみると, リファクタリング前は, bankCharge() への返り値の提供と AccountType.isPremium() の返り値の利用が見られる. 前者に関しては AccountType の種別の増加に伴い同種の結合が増加するため, 全体として関連する結合度は, $nC_{rv}+1$ となる. リファクタリング後は, isPremium() の返り値の利用と, Account.bankCharge() への返り値の提供及び同メソッドからのパラメタの受取りという結合が見られる. isPremium() に関してはリファクタリング前と同様に, AccountType の種別の増加に伴い同種の結合が増加するため, 関連する結合度は, $n+C_{rv}+1$ となる.

この場合リファクタリングの効果は,

$$n(C_{rv}-1)-C_{rv}$$

となる. この値が正になる条件は,

$$n > \frac{C_{rv}}{C_{rv}-1}$$

であり, 例えば $C_{rv} > 1.5$ というかなり保守的な仮定のもとでも, $n \geq 3$ の時にリファクタリング効果が正の値を取る. [2] では「数個の新しいタイプを追加する」場合にこのリファクタリングが有効であるとされており, このリファクタリングに関しては, 我々の結合度による評価は妥当であると言える.

4.2 実験結果と評価

[2] 内の 72 個のリファクタリングについて,

A 適用可能で効果が定量化可能

C 適用可能で場合により効果が定量化可能

B 適用可能で効果が定量化不可能

D 適用不可能

に分類した結果を表 1 に示す.

表 1: 実験結果

A	B	C	D
37	6	15	14

半数以上のリファクタリングに関して, 我々の尺度でリファクタリング効果が定量化可能であることが分かった. 定量化不可能な物の中には, 例えば冗長性を増すことで保守性を高めようとするリファクタリングなどが挙げられ, より包括的な評価のためには, 結合度以外の観点に基づいた尺度も考慮する必要があると考えられる.

5 おわりに

本稿ではリファクタリング効果の定量化について説明した. リファクタリング効果を保守性の改善度ととらえ, 保守性の改善度をプログラム要素間の結合度の低下度合として定量化した.

実際のリファクタリング事例に適用した結果, 我々の提案する手法は半数以上のリファクタリングに対して, その効果を良く定量化できていることを確認した.

今後は別の要因も考慮して, 更に高精度のリファクタリング効果の定量化について研究を進める.

参考文献

[1] K. Beck. *eXtreme Programming eXplained: Embrace Change.* Addison–Wesley, 2000.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison–Wesley, 1999.