Toward Practical Application of Program Refactoring
プログラムリファクタリング技術の実用化に向けた研究

A Dissertation
Submitted to the Graduate School of Information Science and Technology
of Osaka University
in Partial Fulfillment of the Requirements

for the degree

Doctor of Philosophy

in

Information Science and Technology

by

Yoshio Kataoka
大阪大学大学院情報科学研究科
コンピュータサイエンス専攻 D3
片岡 欣夫

2005 年 12 月 21 日

Abstract

プログラムリファクタリングは, プログラムの可読性, 保守性といった品質を向上させるために, その振舞を変える事無く構造のみを改善するための技術である. プログラムリファクタリングの有効性は広く知られているところであるが, 残念ながら実際の開発現場ではあまり使われる事は無い. これには様々な理由が考えられる. 一つにはリファクタリングを適用すべき場所を見つけ出すためにかかるコストの問題が挙げられる. すなわち可読性や保守性といった品質を低下させている要因がプログラム中のどの部分に存在し, 更にはどの様なリファクタリングを適用すれば実際にその問題を解決できるのかといった判断には, 比較的高度な判断やソフトウェア工学に関する知識を必要とする. 別の問題として考えられるのは, リファクタリングの効果, すなわち可読性や保守性の向上度合を定量的に評価するための適当な手段が無いことである. 具体的な効果が見積もられない限り, 特に実際の開発現場では積極的に適用を考える事は難しい.

本研究は, リファクタリング技術の実開発への適用を進めるために, どの様な支援を行うべきかという事に焦点を当てている. 前半部分では, 特定のリファクタリングに対して自動的にリファクタリングを適用すべき箇所を検出する技術の提案を行い, この実用性評価を行った結果を示す. 今回の成果はプログラムの不変情報 (invariant) を利用してリファクタリングの適用箇所を検出することを特徴としている. ただし通常一般のプログラム, 特に現場で開発されているプログラムについては明示的に不変情報が与えられているわけではないので, 不変情報を自動的に検出するためのツールを用いた手法を提案する. 今回用いたのは Daikon と呼ばれるツールで, プログラムの実行履歴から不変情報を自動的に検出する事が出来る. Daikon と不変情報のパターンマッチングを行うツールとを組み合わせる事で, 特定のリファクタリングについて自動的にリファクタリング候補となるプログラム片を検出する事が出来る. 実際に Java で記述されたシステムに対して適用を行い, 実際にリファクタリング候補が検出できる事を確認し, 更には検出したリファクタリング候補が開発者の視点から妥当な物である事を確認した.

後半部分ではプログラムリファクタリングによる保守性向上効果を定量的に評価する手法の提案と評価を行う. 今回の手法ではモジュールの結合度と凝集度に着目し, 保守性向上効果の定量化を行っている. リファクタリングの前後で結合度や凝集度の変化を調べる事で, 保守性の向上度合を定量化する手法であり, 実際に代表的なリファクタリングに対して妥当と思われる定量的評価が行える事を確認した. また結合度ベースの評価に関しては実システムを対象に適用実験を行い, こちらも開発者の視点から妥当な評価結果が得られている事を確認した.

List of Major Publications

1. Yoshio Kataoka, Shinji Kusumoto and Katsuro Inoue, “Supporting Refactoring Using Invariant,” IPSJ Transaction, Vol.46, No.5, pp.1211–1221, May 2005.

2. Yoshio Kataoka, Shinji Kusumoto and Katsuro Inoue, “A Quantitative Evaluation of Refactoring Effect,” Transactions on Information and Systems (Submitted).

3. Yoshio Kataoka, Takeo Imai, Hiroki Andou and Tetsuji Fukaya, “A Quantitative Evaluation of Maintainability Enhancement by Refactoring,” Proceedings of International Conference on Software Maintenance, pp.576–585, Montreal, Canada, October 2002.

4. Takeo Imai, Yoshio Kataoka and Tetsuji Fukaya, “Evaluating Software Maintenance Cost Using Functional Redundancy Metrics,” Proceedings of 26th International Computer Software and Applications Conference, pp.299–306, Oxford, England, August 2002

5. Yoshio Kataoka, Micheal D. Ernst, William G. Griswold and David Notkin, “Automated Support for Program Refactoring using Invariants,” Proceedings of International Conference on Software Maintenance, pp.736–743, Florence, Italy, November 2001

6. Yoshio Kataoka, Masayuki Hirayama, Jiro Okayasu and Tetsuji Fukaya, “An Approach to Reverse Quality Assurance with Data-Oriented Program Analysis,” Proceedings of Asia Pacific Software Engineering Conference 1995, pp.324–332, Brisbane, Australia, December 1995

Toward Practical Application of Program Refactoring
プログラムリファクタリング技術の実用化に向けた研究

大阪大学大学院情報科学研究科
コンピュータサイエンス専攻 D3
片岡 欣夫

ソフトウェアの課題

- ソフトウェア産業の急激な発達に伴う新たな問題(新ソフトウェア危機)
- ソフトウェア工学そのものに訪れた危機(ソフトウェア工学危機)

✓産業の発展にリソースが追いつかない

新ソフトウェア危機

- ソフトウェアの急激な大規模化
  ⇒ 開発量は指数的に増大
- 高機能/多様化により高度な品質保証が困難
  ⇒ 携帯電話の回収騒ぎ
- 機能的品質(機能、性能など)に加え、非機能的品質(保守性、移植性、拡張性など)の向上が必須化
  ⇒ 多様なシリーズ製品の継続的市場投入

非機能要求に対するニーズ

- 携帯電話の回収騒ぎ
- 航空管制システムのトラブル
- 東証システム不具合による市場混乱
- 小田急電鉄の個人情報漏洩

✓何れもソフトの不具合による問題
✓非機能要求の確保が急務になりつつある

非機能要求の例：保守性

- ソフトウェアのライフサイクル
  開発＋保守（不具合修正、機能拡張など）
- 保守性の高いソフトウェアは保守工数の削減に繋がる
- 保守性の高いソフトウェアとは
  - 「良い」設計に基づく
  - 理解性が高い
  - Etc.

リファクタリングとは？

- 「ソフトウェアの振る舞いを変えることなく、理解性や変更容易性の向上を目的として行われる内部構造の変更」
  - M. Fowler “Refactoring”

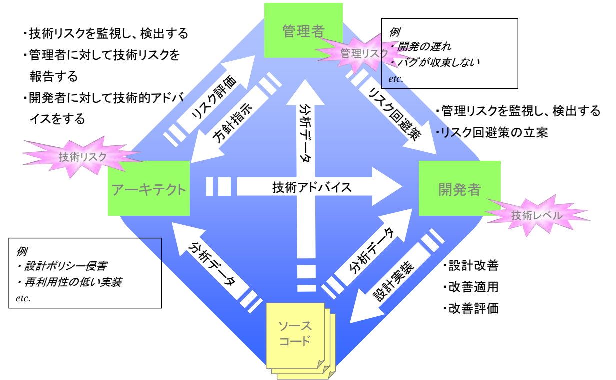
- リファクタリングの効果
  - 設計品質の向上
  - 可読性、理解性の向上
  - 不具合の発見しやすさの向上
  - 開発効率の向上

リファクタリング実適用への課題
～開発者の視点～

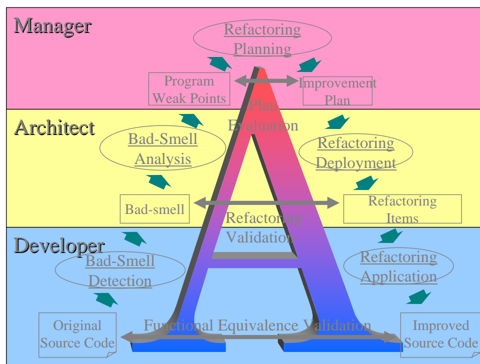
多くのプログラマは

- どこをリファクタリングすべきか分からない
- どの程度有効なのかが分からない
- 新しい不具合を入れ込む危険を冒したくない
- 締め切りに間に合わせるのに精一杯である

リファクタリング実適用への課題
～プロジェクトの視点～



リファクタリングプロセス



三つのサブプロセス

- 改善の立案
  - *リファクタリング候補の検出*
  - *リファクタリング適用効果の見積もり*
  - リファクタリングの優先順位付け
- 改善の実行
  - 改善案の個別リファクタリングへの展開
  - リファクタリングの適用
- 改善の評価
  - 計画評価、*効果評価*、機能不変性の検証

本研究の目的

1. 不変情報を利用したリファクタリング候補検出手法を提案・評価する
2. リファクタリング効果を定量的に見積もり、評価するための手法を提案・評価する

博士論文の内容

1. まえがき
2. リファクタリング概論
3. 不変情報を利用したリファクタリング候補検出に関する研究
   ⇒文献
4. ***結合度・凝集度に着目したリファクタリング効果定量化に関する研究***
   ⇒文献
5. まとめ

結合度・凝集度に着目したリファクタリング効果定量化に関する研究

研究内容

- 定量化手法の検討（結合度と凝集度）
- 結合度計測メトリクスの提案
- 結合度に基づくリファクタリング効果計測事例
- 凝集度計測メトリクスの提案
- 凝集度に基づくリファクタリング効果評価

定量化手法の検討

定量化手法の検討

保守性向上の度合いを効果として定量化

- 適切な保守性メトリクスを選ぶ
- リファクタリング前後で保守性メトリクスの変化を調べる

保守性に関するメトリクス

- 結合度
- 凝集度
- サイズ
- 複雑度
- 冗長度
- 命名規則
- …

結合度・凝集度に着目

- 多くのリファクタリングが結合度の改善を指向している
- 多くの不具合が過度の結合度に由来している
- 比較的定量化が行いやすい
- 凝集度に関しては、結合度を考える上でトレードオフとして考える

結合度/凝集度とは

- Stevens, Myers, Constantine [1]により整理されたソフトウェアの品質メトリクス
- 様々な流儀があるが一般に結合度/凝集度とも7段階のレベルが提案されている
- 結合度
  モジュール間の結合度合いが、どの程度相互依存性を持つかという尺度
- 凝集度
  単一のモジュールに含まれる機能が、どの程度関連を持っているかという尺度

✓ 結合度と凝集度はおおよそトレードオフの関係にある

➢ [1] Stevens, W., Myers, G., Constantine, L.; Structured Design, IBM Systems Journal 13 (2) (1974), 115-139.

結合度計測のメトリクスの提案

結合度評価の観点

- 結合の仕方
  - クラス/インスタンス変数の共有
  - 引数の受け渡し
  - 返り値の受け渡し
- 結合の場所
  - クラス/モジュール内部結合
  - クラス/モジュール外部結合

クラス/インスタンス変数共有 (C_{cv})

変数を共有する事で実現されている結合

e.g. `methodA` が `methodB` で利用されている *c* 個のクラス/インスタンス変数を更新している (or vice versa)

```
class Sample {
  protected int var1, var2;

  public int methodA(int arg1, int arg2) {
    var1 = arg1;
    var2 = arg2;
  }

  private void methodB(void) {
    int temp = var1 - var2;
  }
```

→ $C_{cv} = 2$

引数受け渡し (C_{pp})

引数の受け渡しによって実現されている結合

`methodA` が *p* 個の引数を `methodB` に渡している (or vice versa)

```
class Sample {
  public int methodA(void) {
    methodB(arg1, arg2);
  }

  private void methodB(int arg1, char arg2) {
    ...
  }
}
```

→ $C_{pp} = 2$

返り値受け渡し (C_{rv})

返り値の受け渡しによって実現されている結合

`methodA` が `methodB` の返り値を使っている (or vice versa)

```
class Sample {
  public int methodA(void) {
    char temp = methodB();
  }

  private char methodB(void) {
    ...
  }
}
```

→ $C_{rv} = 1$

クラス内結合とクラス間結合

クラス間結合は、クラス内結合に比べて保守性に与えるインパクトが大きい
→ クラス間結合係数を導入
- K_{cv}：C_{cv}に関する係数
- K_{rp}：C_{pp} 及び C_{rv} に関する係数
- $\mathbf{K_{cv} > K_{rp}}$

結合度メトリクスの定義

- $C_{ci} + C_{ce}$
  - C_{ci}: クラス内結合度
    $C_{cv} + C_{pp} + C_{rv}$
  - C_{ce}: クラス間結合度
    $K_{cv}C_{cv} + K_{rp}(C_{cv} + C_{rv})$

結合度に基づくリファクタリング効果計測事例

結合度を下げるリファクタリング

- 例1：Move Method
  特定のメソッドを適切なクラスに移動する事により、結合度を改善する。
- 例2：Extract Method/Class
  特定の機能を独立したメソッドやクラスとして切り出すことで、結合度を平均化する。

評価の具体例

[Fowler99] 中の例を使って、結合度によってリファクタリング効果を定量化できることを確認する。
- ターゲットプログラミング言語
  Java
- ターゲットリファクタリング
  Move Method

Move Method

- 「あるメソッド (m)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利用されている、あるいは将来的に利用されるようになる。」
  ⇒ もっとも頻繁に利用しているクラスへメソッドを移動する。
- m に関する結合度を、リファクタリング前後で比較
  ⇒ 結合度が減少していれば、リファクタリングは効果的であると言える。

ターゲット
プログラム

銀行口座管理ルーチン
([Fowler99] pp 144)

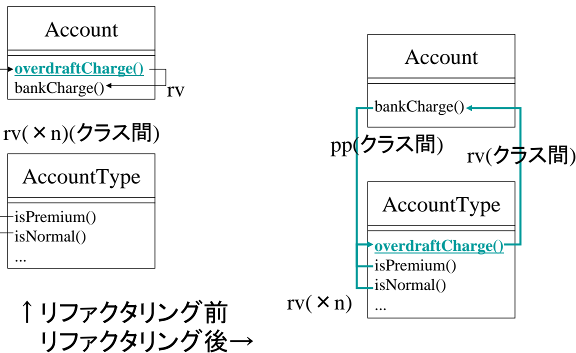
- 口座種別 (`AccountType`) が増加しうる。
- `n` を口座種別 (`AccountType`で扱われる) の数とする。

```
class Account...
  private AccountType _type;
  private int _daysOverdrawn;

  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result += overdraftCharge();
    return result;
  }

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

```
class Account...
  private AccountType _type;
  private int _daysOverdrawn;

  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result += overdraftCharge();
    return result;
  }

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

↑リファクタリング前
リファクタリング後→

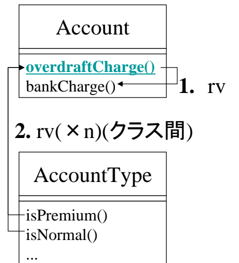
```
class Account...
  private AccountType _type;
  private int _daysOverdrawn;

  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result +=
        _type.overdraftCharge(
          _daysOverdrawn);
    return result;
  }

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;
  }
```



↑リファクタリング前
リファクタリング後→

リファクタリング前

1. `bankCharge()` への返り値の提供
2. `AccountType.isPremium()` の返り値の利用

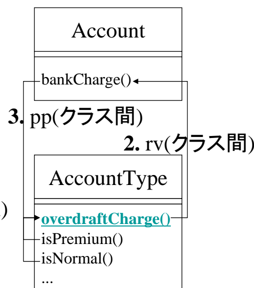


リファクタリング前結合度

$1 + nK_{rp}$

リファクタリング後

1. **`isPremium()`等の返り値の利用**
2. **`Account.bankCharge()`への返り値の提供**
3. **`Account.bankCharge()` からの引数の受け取り**



リファクタリング後結合度

$n + 2K_{rp}$

リファクタリング効果

$$(1 + nK_{rp}) - (n + 2K_{rp}) = \mathbf{n(K_{rp} - 1) - 2K_{rp} + 1}$$

効果が正の値をとるための条件

$$\mathbf{n > 2 + 1/(K_{rp} - 1)}$$

経験的に$K_{rp} > 1.5$ 辺りと仮定すると、
$n > 4$ ならこのリファクタリングは有効と言える

評価例分析

- $K_{rp} > 1.5$
  保守への影響という観点から、返り値や引数による結合に関しては、クラス間結合の方がクラス内結合に比べ、1.5倍以上の開きがある。
- n > 4
  `AccountType` クラスで扱う口座種別が4より大きい場合、Move Methodは効果的。
  → Fowlerの定性的分析と合致 [Fowler99]

リファクタリング効果定量化適用事例

- 社内プロジェクトに、結合度による評価を適用
- 定量化の結果と開発者の主観的評価とを比べる
  - 開発者が有効と判断したものは、定量的にも有効と判断されるか？
  - 開発者が有効でないと判断したものは、定量的にも余り有効でないと判断されるか？

適用対象・条件

- 対象ソフトウェア
  CASEツール (VC++で作成)
- 分析ツール
  リファクタリング支援ツール
- 被験者
  CASEツールの設計・開発者
  (SEとして十年以上のキャリアあり)

実験の手順

1. 結合度の高さが問題になっている部分を特定する*
2. 結合度を測定しておく
3. リファクタリングを適用する
4. リファクタリング結果を定性評価する*
5. 結合度を測り、効果を定量化する
6. 定性的評価と定量的評価を比較する*

※ *: 対象ソフトウェアの開発者が参加

結果

	主観的評価	リファクタリング前	リファクタリング後	効果（差分）
Extract Method(#1)	**可**	10.4	2.8	7.6
Extract Method(#2)	**不可**	12.1	9.0	3.1
Extract Class(#1)	**可**	25.2	9.0	16.2
Extract Method(#3)	**可**	26.4	1.7	24.7
Extract Class(#2)	**良**	126.0	26.0	100.0

考察

- 「不可」の場合
  - 被験者は適用に見合う効果がないと判断された
  - ただし機能拡張を考慮に入れれば、効果的に働く可能性があると認識された
- 「可」の場合
  - 被験者は実際に新規にクラスを切り出そうとしていた
  - 実際に行われたリファクタリングは被験者の想定していたものであった

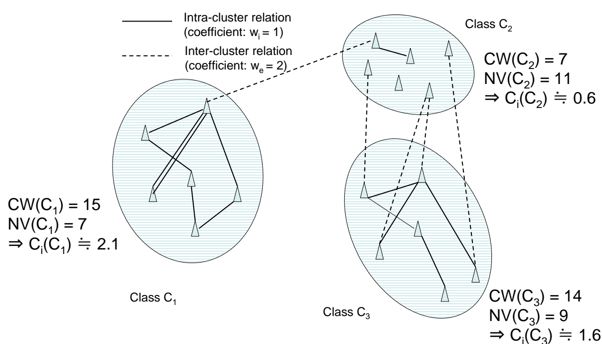
凝集度計測のメトリクスの提案

凝集度(C_i)の定量化

クラスの凝集度の例

- C_i = CW/NV
- CW: クラス内の相互依存度
  クラス内の要素の数と重みで決まる値
- NV: クラス内および外部で接続している
  ノードの数
- クラス内結合加重 (w_i)とクラス間結合加重 (w_e)
  w_i : w_e = 2 : 1 くらいを想定

凝集度の定量化例



凝集度に基づくリファクタリングパターンの評価

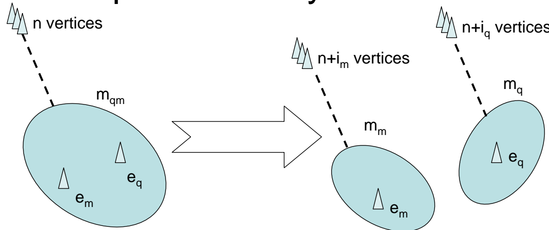
リファクタリングによる凝集度の変化（1）

- Composing Methods: メソッドの凝集度に関係
  - Extract Method: メソッド内から凝集度の高い単位を抜き出すため、メソッドの凝集度を上昇させる (NV–).
  - Inline Method: 上記の逆で、クラスの凝集度を上昇させる (NV–).

リファクタリングによる凝集度の変化（2）

- Simplifying Conditional Expressions: 凝集度には直接関係しないものが多い
- Making Method Calls Simpler: 適用範囲は広いが、効果は限定的
  - Add Parameter: メソッドとクラスの凝集度を上昇させる (CW+, NV).
  - Separate Query from Modifier: 新しく生成されるメソッドの凝集度は上がり、同時にクラスの凝集度も上昇する

例: Separate Query from Modifier



$CW(m_{qm})=nw_e$,
$NV(m_{qm})=2+n$
$\Rightarrow C_i(m_{qm})=nw_e/(2+n)$

$CW(m_m)=(n+im)w_e$,
$NV(m_m)=1+n+i_m$
$\Rightarrow C_i(m_m)=(n+i_m)w_e/(1+n+i_m)$

※ m_mに関して、$n > -2i_m$ の条件でリファクタリング後に凝集度が上昇する
⇒ 右辺は常に負であり、任意のnについて凝集度が上昇することになる
（m_qに関しても同様の議論が成立）

まとめ

- リファクタリングの実用化支援の施策を提案した
  - リファクタリング候補を自動的に検出する技術
  - リファクタリングの効果を定量化する技術
- 提案手法を評価
  - [Fowler99]に紹介されているリファクタリングの一部に関して、検出や効果の定量化が可能である事を確認した
  - 提案方法が設計者の視点から妥当である事を確認した

今後の課題（1）

- 提案手法の高度化
- 提案技術のリファクタリングプロセスへの統合
- リファクタリングプロセス内の他の活動の支援技術の調査及び研究
- 計画から実施まで、包括的なリファクタリング支援環境の構築

今後の課題（2）

リファクタリングの有効性の実地検証

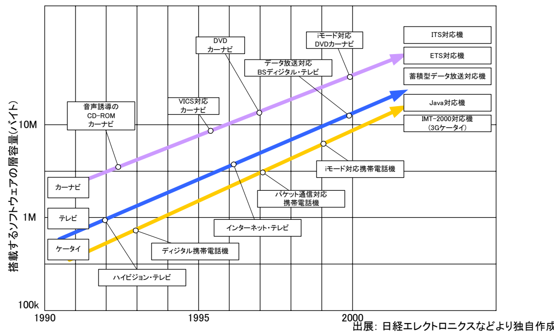
- 不具合分析結果と保守性関連メトリクスとの相関分析
- 利用可能なデータ
  - 不具合数
  - 不具合寿命
  - 不具合密度
  - 不具合発生確率
  - その他

今後の研究者としての歩み

- 企業研究者として
  - 企業でのニーズの発掘
  - 企業からのニーズの発信
  - 技術の実適用の可能性の追求
  - 産学連携の推進
- ソフトウェア工学研究者として
  - リファクタリングに関する技術の深耕
  - ソフトウェア設計力強化に関する技術の調査

以下補足資料

ソフトウェアの大規模化



出展: 日経エレクトロニクスなどより独自作成

ソフトウェアの高機能/多様化
～携帯電話の例～

	FOMA	PDC(NTT)	CDMA2000 1x	cdmaOne	W-CDMA	PDC(VP)
動画+音声	○		○			○
静止画+文字	○	○	○			○
テレビ電話	○				○	
VOD	○		○			
Java/BREW	○	○	○	○		○
LBS	○	○	○	○		○
Web+mail	○	○	○	○	○	○
ローカル通信	○	○				
マルチアクセス	○				○	
国際ローミング				○	○	

LBS: Location Based Services
出展:「移動体通信市場要覧(2003年版)」より独自作成

結合度の分類

- 非直接結合（最弱）
- データ結合
- スタンプ結合
- 制御結合
- 外部結合
- 共通結合
- 内部結合（最強）

凝集度の分類

- 偶然的（最弱）
- 論理的
- 時間的
- 手順的
- 連絡的
- 情報的
- 機能的（最強）

リファクタリングによる凝集度の変化（3）

- Moving Features Between Objects: クラスの凝集度に関連
  - Move Method, Move Field: クラスの凝集度を上昇させる (NV-)
  - Extract Class: 新しく得られたクラスの凝集度と、残されたクラスの凝集度が高まる

リファクタリングによる凝集度の変化（4）

- Organizing Data: データ構造に関するリファクタリング
  - Self Encapsulate Field: クラスの凝集度を減少させる (NV+)
  - Replace Array with Object: クラスの凝集度を上昇させる

リファクタリングによる凝集度の変化（5）

- Dealing with Generalization: 凝集度の変化を評価することは難しい
  - Pull Up Method, etc.: コードの冗長性を削除するが、サブクラスの凝集度は（値としては）下がり(CW-, NV-1)、親クラスの凝集度は上がる(CW+, NV+1).
  - Push Down Method, etc: サブクラスの凝集度を上げ、親クラスの凝集度を上げる
  - Extract Subclass: Extract Classと事情は同様

リファクタリングによる凝集度の変化（6）

- Big Refactorings: 戦略的リファクタリングで、凝集度に対する影響も大きいが、一様に議論することは難しい
  - Tease Apart Inheritance: クラス階層を非常に凝集度の高い小さなクラス階層に変換する戦略
  - Convert Procedural Design into Objects: 新しく得られるクラスは凝集度が高くなる
  - Extract Hierarchy: 新しく得られるクラスは凝集度が高くなる