**修士学位論文**

題目

# CK メトリクスを用いてリファクタリングの効果を予測する手法の提案

指導教員
井上 克郎 教授

報告者
松本 義弘

平成 19 年 2 月 13 日

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 ソフトウェア工学講座

平成 18 年度 修士学位論文

# CK メトリクスを用いてリファクタリングの効果を予測する手法の提案

松本 義弘

## 内容梗概

ソフトウェアの保守性を改善する手段としてリファクタリングが挙げられている．リファクタリングとは，ソフトウェアの外部的な振る舞いを保ったままで，内部構造を改善する技術である．これまでに，リファクタリングの位置特定手法，リファクタリングパターンの選択手法に関する研究が行われている．特に構造的な欠陥のあるクラスを特定し，それに対して有効なリファクタリングを提供している．一方，実際の開発現場では，構造的な欠陥のある箇所に限らず，機能的な観点からリファクタリングを行い，保守性の改善を試みることが多々ある．そのため，開発者は，あらゆる場面で適用するリファクタリングの影響範囲を把握し，保守性に与える効果を評価する必要がある．本稿では，ソースコードを修正する前にリファクタリングがソフトウェア保守に与える効果を予測する手法を提案・実装し，その手法の有用性を評価する．具体的には，ソフトウェアの複雑さを評価するために用いられる CK メトリクスを用いて，リファクタリングが保守性に与える効果を予測する手法を提案する．さらに，ユーザが抱える問題を解消するリファクタリング候補をいくつか挙げ，それらに対しソフトウェア保守に与える効果を予測する適用実験を行った．そこで，ユーザが最初に主観で採用していたリファクタリング候補よりもよい候補を提示することができ，本手法の有用性を確認することができた．

## 主な用語

リファクタリング (Refactoring)
CK メトリクス (CK Metrics)
ソフトウェア保守 (Software Maintenance)

# 目次

**1 まえがき** **4**

**2 ソフトウェア保守とリファクタリング** **6**

2.1 ソフトウェア保守 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.2 リファクタリング . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2.1 リファクタリングの定義とプロセス . . . . . . . . . . . . . . . . 7

2.2.2 リファクタリングの効果予測とその問題点 . . . . . . . . . . . . . . 10

2.3 ソフトウェアの保守性とソフトウェア複雑度メトリクス . . . . . . . . . . . . 11

**3 CK メトリクスを用いたリファクタリングの効果予測手法** **14**

3.1 対象プログラムの解析および CK メトリクス計測 (STEP1) . . . . . . . . . . . 15

3.2 リファクタリング箇所とリファクタリングパターンの入力 (STEP2) . . . . . . 15

3.3 解析情報の修正 (STEP3) . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.3.1 リファクタリングの影響を受けるクラスの検出 . . . . . . . . . . . . 16

3.3.2 ユーザ非依存部分の修正 . . . . . . . . . . . . . . . . . . . . . . . 17

3.3.3 ユーザ依存部分の修正 . . . . . . . . . . . . . . . . . . . . . . . . 18

3.4 CK メトリクスの再計算 (STEP4) . . . . . . . . . . . . . . . . . . . . . . 19

**4 実装** **20**

4.1 入出力部 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4.2 プログラム解析部 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

4.3 解析情報修正部 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.4 CK メトリクス計測部 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

**5 評価** **27**

5.1 実験対象プログラムの概要 . . . . . . . . . . . . . . . . . . . . . . . . . . 27

5.2 実験概要 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

5.3 実験結果 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

5.4 考察 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**6 関連研究** **33**

6.1 リファクタリングの効果計測手法 . . . . . . . . . . . . . . . . . . . . . . . 33

6.2 リファクタリングの適用支援ツール . . . . . . . . . . . . . . . . . . . . . . 33

**7 まとめ** **34**

**謝辞** **35**

**参考文献** **36**

**付録** **39**

**A リファクタリングパターン** **40**

A.1 フィールドの移動 . . . . . 40
A.2 フィールドの引き上げ . . . . . 40
A.3 フィールドの引き下げ . . . . . 41
A.4 メソッドの移動 . . . . . 41
A.5 メソッドの引き上げ . . . . . 41
A.6 メソッドの引き下げ . . . . . 42
A.7 クラスの抽出 . . . . . 42
A.8 スーパークラスの抽出 . . . . . 43
A.9 サブクラスの抽出 . . . . . 43

# 1 まえがき

近年，ソフトウェアの大規模化・複雑化に伴い，ソフトウェアの保守作業がますます困難になってきている．ソフトウェア保守では，フィールドバグの修正，環境変化に対する機能追加・変更， 将来トラブルにつながりそうな箇所に対する対応等の作業が実施される [18]. しかし，保守対象のソフトウェアがうまく設計されていない，度重なる変更により構造がわかりにくくなってしまう，変更履歴のドキュメントが存在しない等の問題により，ソフトウェア保守コストは増大してきている．実際に，多くのソフトウェア会社が既存システムの保守に非常に多くの人的，時間的コストをかけていると報告されている [26].

ソフトウェアの保守性を改善する手段としてリファクタリング [13] が挙げられる．リファクタリングとは，ソフトウェアの外部的振る舞いを保ったまま内部の構造を改善する技術である．様々な開発現場でリファクタリングが行われており，その有用性は認められている．しかし，リファクタリングを行う箇所や適用するリファクタリングが適切でないと，適用コストを上回る効果が期待できない．そこで，これまで多くのリファクタリング位置特定手法およびリファクタリングパターン選択法が提案されてきた [4][27][23]. これらの手法で共通している点は，構造的な欠陥のあるクラス（例えば，低凝集度や高結合度のクラス）を特定すること，そして，その欠陥を解消するために有効なリファクタリングパターンを提供しているということである．一方で，実際の開発現場では，構造的な欠陥の解消に限らず，機能的な観点からリファクタリングを行い，保守性の改善を試みることが多々ある．例えば，“ 計算処理を行うクラスに画面描画の機能の一部が実装されているので，この機能を画面描画のクラスに移動したい ”，がその例である．開発者は，あらゆるケースにおいて，リファクタリングパターンの適用前に影響範囲を特定し，リファクタリングが保守性に与える効果を評価する必要がある．

本稿では，ソースコードを修正する前にリファクタリングがソフトウェア保守に与える効果を予測する手法を提案し，その有用性を評価する．具体的には，リファクタリングの保守性を評価するために CK メトリクスを導入し，メトリクスの計測に必要な結合・継承関係とクラス内部複雑度を対象プログラムから解析する．そして，開発者が与えたリファクタリングパターンを解析情報に反映させることでソースコードを修正せずに CK メトリクスを計測することを可能にする． さらに，本手法の有用性を確認するために，ユーザが抱える問題を解消するリファクタリング候補をいくつか挙げ，それらに対しソフトウェア保守に与える効果を予測する適用実験を行う．

以降，2 章では，ソフトウェア保守とリファクタリングに関して述べ，リファクタリングの効果予測に関する問題点を挙げる．3 章では CK メトリクスを用いたリファクタリングの効果予測手法を提案し，4 章では，提案手法の具体的な実装方法について述べる，5 章では，

本研究室で開発中のプログラムを対象に適用実験を行い，適用実験の結果と本ツールの有用性の評価を行う．6 章では，関連研究について述べ，最後の 7 章では，まとめと今後の課題を述べる．

# 2　ソフトウェア保守とリファクタリング

## 2.1　ソフトウェア保守

ソフトウェア保守とは, “ 納入後, ソフトウェア・プロダクトに対して加えられる, フォールト修正, 性能または他の性質改善, 変更された環境に対するプロダクトの適応のための改訂 ” であると定義されている [1]. また, 目的毎に次の 4 つのカテゴリに分けられている.

**修正のための保守 (Correction)** 発見された問題を修正するために, 納入後に実施される, ソフトウェア・プロダクトの対処的改変,

**適応のための保守 (Adaptation)** 変化した, または変化しつつある環境において, ソフトウェア・プロダクトを続けて使用可能なように維持するために, 納入後に実施される, ソフトウェア・プロダクトの改変,

**完全化のための保守 (Perfection)** 性能または保守性を改善するため, 納入後に実施される, ソフトウェア・プロダクトの改変,

**予防のための保守 (Prevention)** ソフトウェア・プロダクトのなかに潜む, 潜在的なフォールトが, 効果的なフォールトに転じる前に, それを検出し, 修正するために, 納入後に実施される, ソフトウェア・プロダクトの改変.

ソフトウェア保守は, ソフトウェアライフサイクルコストの大きな部分を占めており [12], ライフサイクルを考えるとき, その費用と労力からみて保守は非常に大切な工程である. しかし, ソフトウェア保守における課題は多くあり, Dorfman および Thayer[10] は, 保守に関しては, 投資効果が明らかにならないので, 常に資源を獲得するための戦いが起きると述べている. 資源を巡って争うということが常に存在し, 次のソフトウェア・プロダクトに対するコードを作成しながら, 将来の納入計画を決め, 現在のソフトウェア・プロダクトに対して緊急的なパッチを施すということは難題である. また, ソフトウェアを開発したチームは, ソフトウェアが運用に入ると必ずしも保守を担当するとは限らない. 自分の開発したものではない, 大規模なソースコードに潜む欠陥を発見しなければならないということは, 保守者にとっては非常に難題である.

多くの場合, 独立したチームが, ソフトウェアが適切に運用されユーザの変化するニーズを満たすように進化させられていることを確実にするために雇用されているが, 保守のアウトソーシング (社外調達) も, 主要な産業になりつつある. 大企業は, 非公開としたいビジネス中核となるソフトウェアを除いては, ソフトウェア保守を含めて, 運用をアウトソーシングする. このような背景のために, 開発者は通常コードを説明しなければならない場には

居合わせないことが多く，変更も文書化されていないことも多い．したがって，保守者は，ソフトウェアに関して制限された理解しか得ることができず，ソースコードを自分で読まねばならない．文献 [18] では保守作業のおおよそ 40%から 60%は，改変すべきソフトウェアの理解に費やされていると指摘されている．したがって，保守作業の効率を高めること，さらにプログラムの理解容易性を高めるということは，ソフトウェア工学における重要な課題の 1 つとなっている．

## 2.2 リファクタリング

### 2.2.1 リファクタリングの定義とプロセス

リファクタリングとは“ 外部からみたときの振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を変化させること ” であると定義されている [13]．ソフトウェアは，開発者によるコードの変更が短期的な視野にたったものであったり，設計の全体的な理解をせずに行われたりすると，コードは構造を崩し，コードを読んで構造を把握することも難しくなる．また，設計が把握しづらくなるにつれ，それを維持するのが困難になり品質の劣化を招くこととなる．それに対して，リファクタリングを行うことにより，ソフトウェア設計品質を向上させ，ソフトウェアは理解しやすくなる．同時に，コードが理解できるようになると，プログラムが何を行っているのかが明確になりバグを見つけやすくなる．次に，リファクタリングのプロセス [20] について紹介する．

STEP1:どこにリファクタリングを適用するかを決定する

STEP2:どのようなリファクタリングを適用するかを決定する

STEP3:リファクタリングの効果を予測する

STEP4:プログラムの修正を行う

STEP5:修正後のソースコードのテストを行う

このリファクタリングプロセスについて，段階ごとに詳しく説明する．STEP1 では，まずソフトウェア品質を劣化させ，将来的にソフトウェアの保守コストを増大させると思われるコードを見つける．STEP2 で，そのコードが抱える問題を解消するために効果的なリファクタリングを選択する．Fowler[13] は，リファクタリングすべき問題とそれを解消する有効なリファクタリングパターンをまとめている．具体的な例として，“ たくさんの役割を担い，フィールドやメソッドが多すぎるクラスに対しては，クラスの抽出やフィールド，メソッドの移動を適用し，役割を分散すべきである ”が挙げられる．STEP3 では，決定したリファク

タリングがソフトウェアの保守性に与える効果を見積もり，将来的な保守作業がどれだけ軽減されるかを把握する．STEP4,5 では，実際にプログラムの修正を行い，修正したプログラムがリファクタリング前のプログラムの動作と変化がないかテストを行う．以上のような段階を踏むことで，リファクタリングは適切に実施される．次に，主要なリファクタリングパターンをいくつか紹介する．

メソッドの抽出

ひとまとめにできるコード片がある場合に，新たなメソッドとして定義し，抽出されたコードを抽出先のメソッドへの呼び出し文に置き換える (図 1 参照).

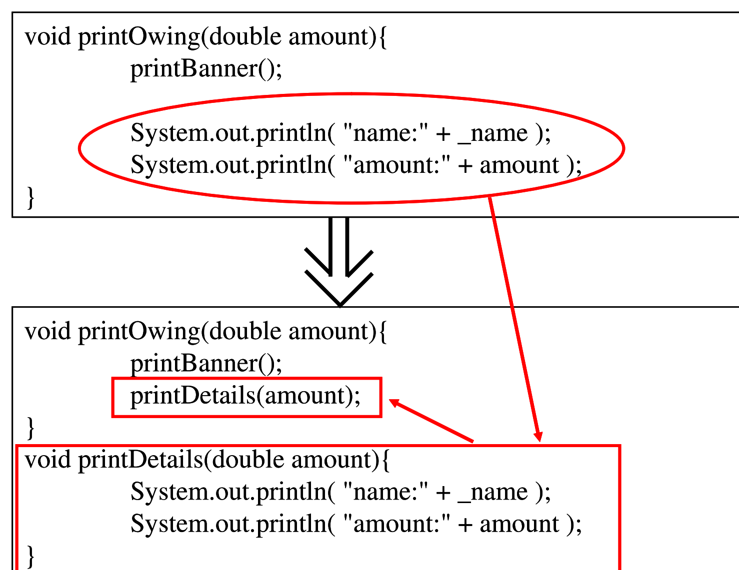



図 1: リファクタリング (メソッドの抽出)

メソッドの引き上げ

同じ結果をもたらすメソッドが複数のサブクラスに存在した場合，それらを親クラスに引き上げる．最も単純なケースは，複数のメソッド本体が全く同じである場合である (図 24 参照)．重複したコードが兄弟クラスに存在した場合には，メソッドの抽出を行ってから，メソッドの引き上げを行えばよい．

クラスの抽出

2 つのクラスでなされるべき作業を 1 つのクラスで行っている際に，新たにクラスを作って，適当なフィールドとメソッドを元のクラスからそこに移動する (図 26 参照).

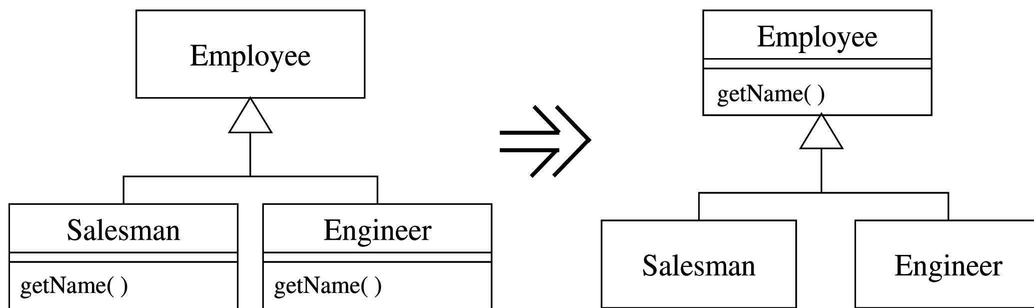



図 2: リファクタリング (メソッドの引き上げ)

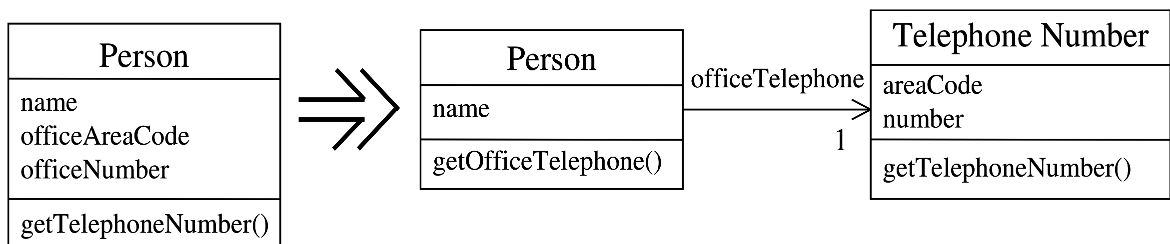



図 3: リファクタリング (クラスの抽出)

さらに，Fowler[13] はリファクタリングを行う理由として，以下の 4 つを挙げている．

リファクタリングはソフトウェア設計を向上させる

ソフトウェアは，開発者によるコードの変更が短期的な視野にたったものであったり，設計の全体的な理解をせずに行われたりすると，コードは徐々に構造を崩すことになる．さらに，コードの構造が悪化すると累積的に悪影響が及び，コードを読んで設計を把握することも難しくなる．開発者はリファクタリングを定期的に行うことで，コードの構造悪化を未然に防ぎ，良質な状態を保つことができる．

リファクタリングはソフトウェアを理解しやすくする

ソフトウェアは，何かの変更を加えるためにコードを書いた本人とは異なる人がコードを読む可能性がある．この利用者の存在は軽視されがちであるが，実際には非常に重要である．開発過程において，少しの時間をリファクタリングに充てるだけで，コードの目的がより伝わるようになる．また，リファクタリングはコードの不明な部分を理解するにも用いることができる．例えば，コードを理解する際，何をしているのかを突き止めるのに自分の理解をよりよい形で反映するように実際にコードを書き換え

る．そして，テスト結果が変更前と変わらないことで，自分の理解が間違っていなかったことを確かめることができる．

リファクタリングはバグを見つけ出す

コードが理解できるようになると，バグが見つけやすくなる．プログラム構造を明確にすることで，コードに対する推測が正しかったとわかり，無理なくバグを発見できる．また，バグレポート自体をリファクタリングが必要な兆候と考えることもできる．コードが不明確なためバグを発見できなかったことを示しているからである．

リファクタリングでより速くプログラミングできる

優れた設計は，ソフトウェア開発のスピードを一定に保つのに役立つ．リファクタリングは，設計を改善しコードを理解しやすくすることで，従来に比べて開発期間の短縮につながる．

以上のように，リファクタリングは,“ 単なるコードのクリーニング作業 ”という意味にとどまらず，バグフィクスや機能追加，コードレビューなど作業全体のあらゆる場面で有用であると言える．

### 2.2.2　リファクタリングの効果予測とその問題点

リファクタリングは，ソフトウェアの保守性を高める有効な作業である．しかし，リファクタリングを行う箇所や適用するリファクタリングが適切でないと，適用にかかるコストを上回る効果が期待できない．不適切なリファクタリングによって，保守性を悪化してしまう危険性も考えられる．一方，開発者はリファクタリングで設計上の問題を解消するのに，保守性がどの程度改善したかを見積もるのは容易ではないのが実状である．また，実際の開発現場では，特にリファクタリングの効果予測の要求が高いケースがある．次に，その問題例をあげる．

“ ある GUI の処理とは無関係なクラスに，GUI の機能が誤って実装されている ”という問題に対し,“ その機能を GUI の処理を行うクラスに移動する ”というリファクタリングを行うとする．その際，GUI の処理を行うクラスが複数あり，どのクラスに移動しても意味的には正しい場合に将来のソフトウェア保守に対してどのクラスに移動するのが効果的であるかを判断する必要がある．

この状況において，開発現場では開発者の直感でどのクラスに適用するかを決定してしまうことが多い．そのため，選択したリファクタリングは，開発者の一時的な理解のしやすさは改善するが構造的には悪化するものであり，その後の保守作業がリファクタリング前より困難になっている可能性がある．このような場合，コードの修正やテストにかけたコストが

無駄になるだけでなく，リファクタリング前のコードに修正し直すコストも必要になる．この問題を解消するためには，ソースコードを修正する前にリファクタリングがソフトウェア保守に与える効果を予測する定量的な指標が必要であると考える．

### 2.3　ソフトウェアの保守性とソフトウェア複雑度メトリクス

前節でリファクタリングによってソフトウェアの保守作業の効率を高めること，プログラムの理解容易性を高めることの重要性を述べた．さらに，ユーザが将来的なソフトウェアの保守コストを軽減する適切なリファクタリングを行うため，効果を予測する必要があることについても言及した．

ソフトウェア保守の容易さを評価する一指標として，ソフトウェア複雑度メトリクスがある．ソフトウェア複雑度メトリクスの測定結果が高い，つまり，複雑であればあるほど修正コストがかかり，保守が困難であると評価されている [28]．これまでに提案された代表的なソフトウェア複雑度メトリクスには，Halstead のメトリクス [14]，McCabe のサイクロマチック数 [19] などがある．Chidamber と Kemerer は，これらのメトリクスは従来の (非オブジェクト指向の) プログラミング言語で開発されたソフトウェアに対する複雑度メトリクスであり，オブジェクト指向設計を用いて開発されたソフトウェアの複雑度を評価するには不十分であると指摘し，オブジェクト指向設計で開発されたソフトウェアを対象とする 6 つの複雑度メトリクスを提案した [14].

CK メトリクスは，クラスの構造に基づいて，その複雑度を静的に評価する．CK メトリクスは，Weyuker が提案した複雑度メトリクスが満たすべき数学的性質 [25] をおおむね満たすことが確認されている [6]．CK メトリクスはまた，複数の実験によって，エラーの発生を予測する精度が評価されている [3][15][7]．CK メトリクスの変種も多数提案されており，文献 [5] では，CK メトリクスおよびその変種を含む多くの複雑度メトリクスを同一のデータによって比較している．特定のプログラミング言語で記述されたソースプログラムから CK メトリクスを抽出するツールも開発されている [21]．以下に，文献 [3] からの引用した，CK メトリクスの定義を示す.

**WMC(Weighted Methods per Class)**：

計測対象クラス C が，メソッド $M_1,\cdots,M_n$ を持つとする．これらのメソッドの複雑度をそれぞれ $c_1,\cdots,c_n$ とする．このとき，$WMC(C) = \sum c_i$ である．メソッドの複雑度の具体的な算出方法は述べられていないが，Halstead のメトリクス [14]，McCabe のサイクロマチック数 [19] などを用いる方法が考えられる．

**DIT(Depth of Inheritance Tree)**：

```
int method(int a, int b){
  int c, d;

 d=0;
 if(a >= b){
   c = a / b;
  } else{
   c = b / a;
  }
  while(c != d){
   d++;
    a = a * b ;
  }
  return a;
}
```

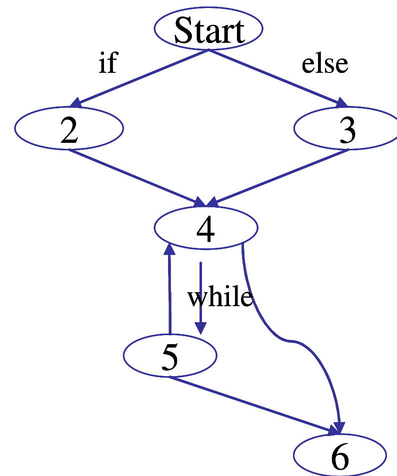



図 4: Halstead のメソッド複雑度の計算例

DIT は計測対象のクラスの継承木内での深さである．多重継承が許される場合は，DIT は継承木におけるそのクラス (を表す節点) からそれ以上親クラスが存在しないクラス (根) に至る最長パスの長さとなる．

**NOC(Number Of Children)** ：

NOC は計測対象クラスから直接導出されているサブクラスの数である．

**CBO(Coupling Between Object classes)** ：

CBO は，計測対象クラスが結合しているクラスの数である．あるクラスが他のクラスのメソッドやインスタンス変数を参照しているとき，結合しているという．

**RFC(Response For a Class)** ：

計測対象のクラスのメソッドと，それらのメソッドから呼び出されるメソッドの数の和として定義される (すなわち，メッセージに反応して潜在的に実行されるメッセージの数となる)．

**LCOM(Lack of Cohesion in Methods)** ：

計測対象クラス $C_i$ が $n$ 個のメソッド $M_1$,…,$M_n$ を持つとする．$I_i (i = 1, \cdots, n)$ をそれぞれメソッド $M_i$ によって用いられるインスタンス変数の集合とする．$P = \{(I_i, I_j)|I_i \cap I_j = \phi\}$ と定義し，$Q = \{(I_i, I_j)|I_i \cap I_j \neq \phi\}$ と定義する．もし $I_1$,…,$I_n$ がすべて $\phi$

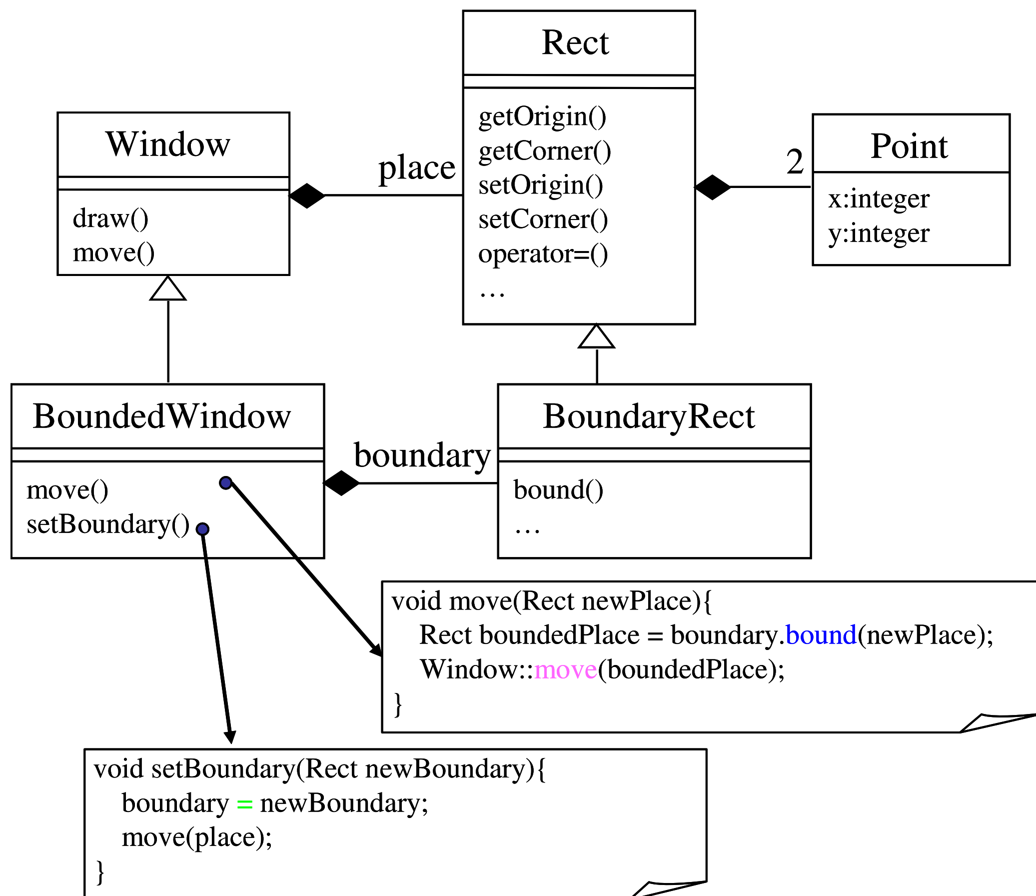



図 5: CBO と RFC の計測方法を説明するための図

の時は，$P = \phi$ とする．このとき，$LCOM = |P| - |Q|$，ただし，値が 0 より小さくなるときは 0，と定義する．

いずれのメトリクスも，計測値は 0 以上になり，計測対象のクラスが複雑になるほど，その計測値が大きくなる．図 5 は CBO と RFC の計測方法を示すための例である．図中のクラス BoundedWindow は，クラス Window から導出され，インスタンス変数 boundary（型はクラス BoudaryRect），メソッド move() と setBoudary() を持っている．メソッド move() の定義中で，クラス BoundaryRect のメソッド bound() と，クラス Window のメソッド move() を呼び出している．メソッド setBoudary() の定義中で，クラス Rect のメソッド operator=() を呼び出している．BoundedWindow は 2 つのメソッドを持っており，他のクラスの 3 つのメソッドを参照しているので，RFC は 5 となる．BoundedWindow は，クラス BoundaryRect, Rect, Window の 3 つのクラスを（いずれもメソッド呼び出しによって）参照しているので CBO は 3 となる．

# 3　CK メトリクスを用いたリファクタリングの効果予測手法

本節では，CK メトリクスを用いてソフトウェア保守に与える効果を予測する手法を提案する．本手法は，オブジェクト指向ソフトウェアに対して，以下 4 つの STEP を通じて CK メトリクスを計測する．また，各 STEP で行う処理を図 6 に示し，提案手法の大まかな流れを説明する．

STEP1：対象プログラムの解析および CK メトリクス計測

STEP2：リファクタリング箇所とリファクタリングパターンの入力

STEP3：解析情報の修正

STEP4：リファクタリング前後のメトリクスの変化率

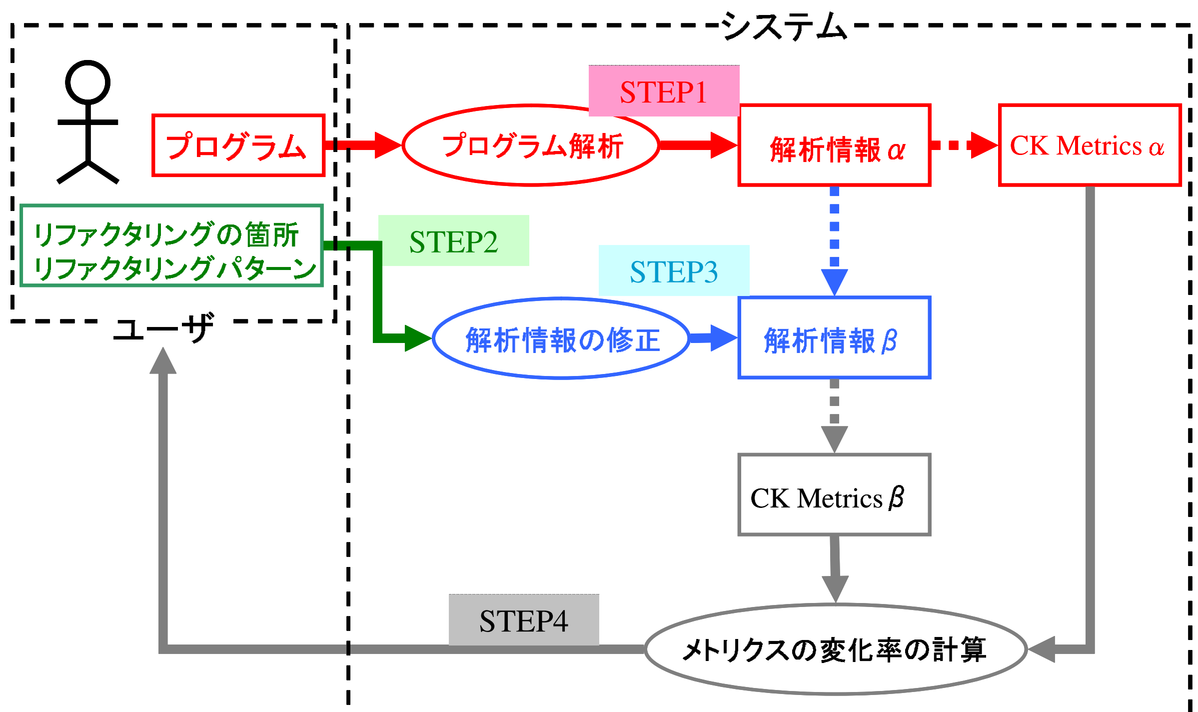



図 6: 提案手法の概要

ここで，STEP2 はユーザが行う手続きであり，STEP3 は場合によってユーザの入力を必要とする．以下では，これら 4 つの STEP を 1 つずつ詳しく説明していく．

### 3.1 対象プログラムの解析および CK メトリクス計測 (STEP1)

ユーザによって与えられたリファクタリング対象のプログラムを解析し，CK メトリクスの計測を行う．対象プログラムを解析し，取得する必要のある情報は，以下のとおりである．

- クラス間の参照関係

  あるクラスがフィールドまたはメソッドによって，他のクラスを参照しているかを調べ，すべてのクラスに対し，参照・被参照なクラスを取得する．

- クラス間の継承関係

  あるクラスが他のクラスを継承している，もしくは，継承されているかどうかを取得する．

- メソッド間の参照関係

  あるメソッドが他のメソッドを参照している (Fan-out)，もしくは，参照されている (Fan-in) 関係を取得する．その際，どのインスタンスに対して呼び出すのか，あるいは，静的な呼び出しか，などの情報も合わせて取得する．

- フィールド，メソッドの情報とそれらの参照関係

  あるクラスの各フィールドが，どのメソッドから参照されているかを取得する．

- メソッドの複雑さ（サイクロマチック数）

  各クラスの全メソッドに対して，それらの繰り返し文や分岐の構造を解析し，それを基にメソッドの複雑さを取得する．また，メソッドの複雑さは，サイクロマチック数 [19] を用いる．

CBO,NOC,DIT の計算はクラス間の参照関係や継承関係，RFC はメソッド間の参照関係から計算される．WMC は，メソッドの複雑さ，LCOM はフィールドとメソッドの間の参照関係から計算する．

### 3.2 リファクタリング箇所とリファクタリングパターンの入力 (STEP2)

ユーザは，リファクタリングを行う箇所（フィールド，メソッド，クラス）を指定し，適用するリファクタリングパターンを与える．本手法では，構造的な変化を伴うリファクタリングパターンのみを対象とし，その中の一部を実装した．本稿で対象にしているリファクタリングパターンは以下の通りである．

- フィールドの移動

- フィールドの引き上げ
- フィールドの引き下げ
- メソッドの移動
- メソッドの引き上げ
- メソッドの引き下げ
- クラスの抽出
- スーパークラスの抽出
- サブクラスの抽出

今回，実装したリファクタリングパターンは，フィールド，メソッドといったあるまとまった単位のものに対して，適用できるものに限定した．つまり, “ メソッドの抽出 ”のように, あるメソッド内の一部分を新たにメソッドとして作成するといったリファクタリングパターンは対象外とした.

### 3.3　解析情報の修正 (STEP3)

3.2 節で与えられたリファクタリング箇所とリファクタリングパターンを元に，3.1 節で取得した解析情報に対して修正を行う．本節では，クラス A のメソッド a1() をクラス B に“ メソッドの移動 ”を適用する例 (図 7) を挙げ，具体的に解析情報の修正方法について説明する．ここで，クラス A，B と，メソッド a1() の呼び出し関係を図 7(a) のような関係であると仮定し，説明を行う.

#### 3.3.1　リファクタリングの影響を受けるクラスの検出

リファクタリングは，リファクタリング対象のクラス (例の場合，移動元のクラス A と移動先のクラス B) に対してのみ，修正すれば良いとは限らない．まず初めに，リファクタリング対象クラス以外で影響を受けるクラス群をすべて検出する．このリファクタリングは, メソッド a1() の移動を行うため，メソッド a1() を参照しているメソッドは参照先を変更する必要がある．そのため，それらのメソッドが属するクラスはすべて影響を受けるクラスとなる．図 7(a) では，メソッド a1() を参照しているメソッド b1(),c1(),d1() が参照しているため，これらのメソッドが属するクラス B,C,D はすべて影響を受けるクラスとして検出される．具体的には，3.1 節で取得した解析情報の中で，あるメソッドが他のどのメソッドから参照されているか (Fan-in) の情報を利用して，メソッド b1(),c1(),d1() は検出される.

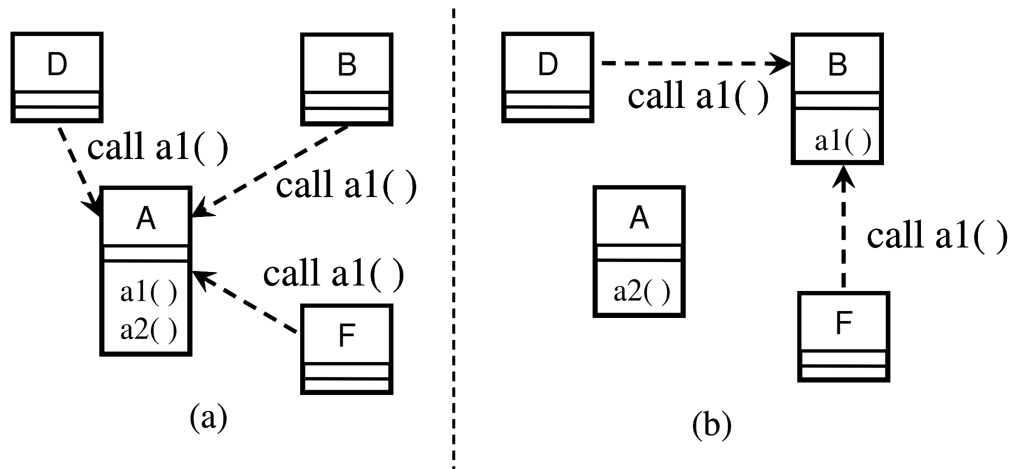



図 7: クラス A から B へのメソッドの移動

#### 3.3.2 ユーザ非依存部分の修正

次に，解析情報の修正を行う．解析情報の修正は，“ ユーザ非依存部分の修正 ”と“ ユーザ依存部分の修正 ”の 2 段階に分かられる．本節では，“ ユーザ非依存部分の修正 ”について述べる．ユーザ非依存部分の修正は，システムが機械的に変換できる処理に限られる．機械的に変換できる処理とは，“ 解析情報の修正がユーザに依存せず，一意に変換が決まる処理 ”のことを指す．例えば，図 8 のように，メソッド a1() は，クラス A に属しているという情報をクラス B に移動する，また，クラス B から A のメソッド a1() への参照をクラス B 自身の参照に変換するといった処理が挙げられる．

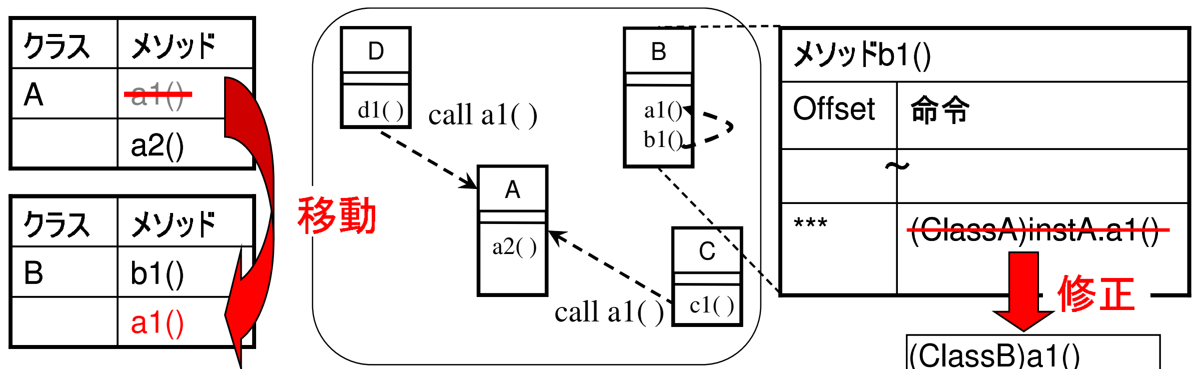



図 8: クラス A から B へのメソッドの移動 (2)

### 3.3.3　ユーザ依存部分の修正

前項で，ユーザ非依存部分の修正だけではなく，ユーザ依存で修正する必要がある箇所が発生する場合がある．例えば，図 9 では，メソッド a1() の移動後の参照関係を示しているが，クラス D はクラス B のメソッド a1() をどのインスタンスに対して呼び出すのかを決定する必要がある．その際，ユーザに入力を求め，得た情報を元にどのインスタンスに対して呼び出すのかを決定する．“ メソッドの移動 ”に関しては，メソッド a1() を参照しているメソッドとして検出された c1(),d1() に対し，a1() を呼び出している命令を検索することで，ユーザ依存箇所を検出している．

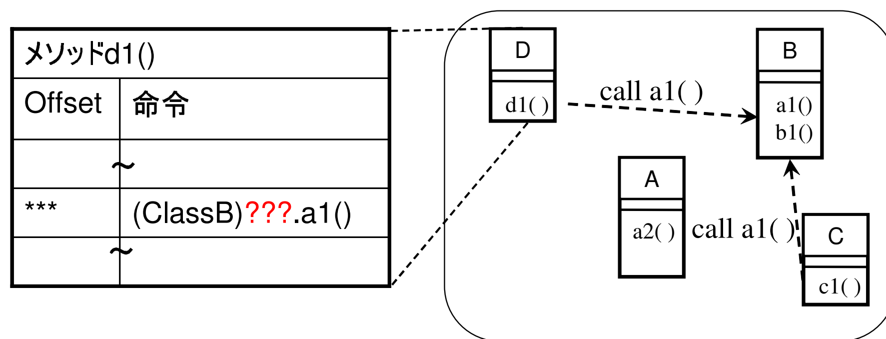



図 9: クラス A から B へのメソッドの移動 (3)

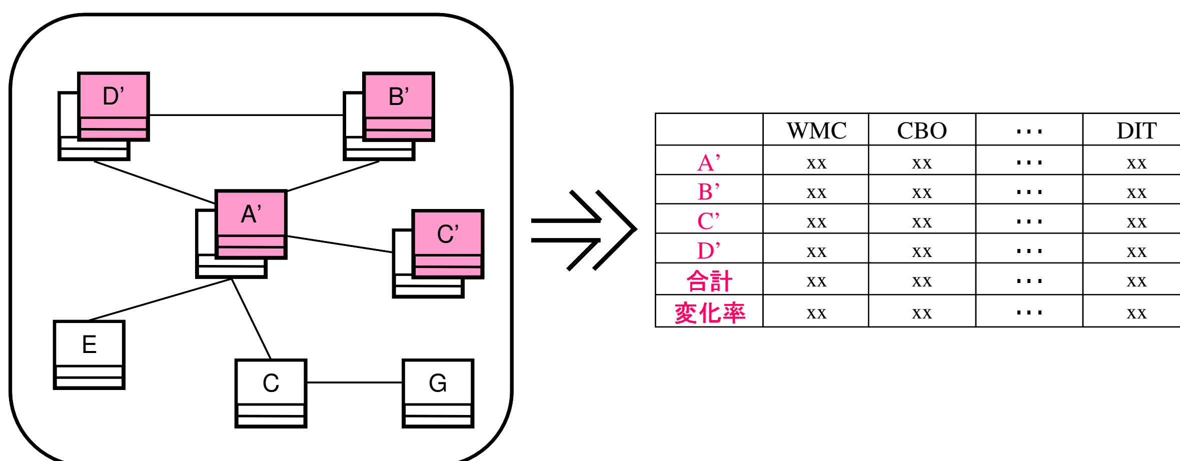
### 3.4 CK メトリクスの再計算 (STEP4)

最後に，3.1 節と同様に，修正された解析情報からリファクタリング後の CK メトリクスを計測する．そして，CK メトリクスの変化があったクラスに対し，その各メトリクスの合計と，3.1 節で計測した CK メトリクスを元にリファクタリング前後の変化率を計測し，それをユーザにフィードバックとして与える．

図 10 の例では，クラス A,B,C,D が CK メトリクスの変化があったクラスと検出され，それぞれのメトリクスの変化率を計測する．WMC の変化率は以下に示す．

$$\frac{SUMwmc(A',B',C',D') - SUMwmc(A,B,C,D)}{SUMwmc(A,B,C,D)}$$

SUMwmc(A,B,C,D) は，クラス ABCD に対するリファクタリング前の WMC の合計を表す．同様に，SUMwmc(A’,B’,C’,D’) はクラス ABCD に対するリファクタリング後の WMC の合計を表す．




| | WMC | CBO | ··· | DIT |
|---|---|---|---|---|
| A’ | xx | xx | ··· | xx |
| B’ | xx | xx | ··· | xx |
| C’ | xx | xx | ··· | xx |
| D’ | xx | xx | ··· | xx |
| 合計 | xx | xx | ··· | xx |
| 変化率 | xx | xx | ··· | xx |

図 10: クラス A から B へのメソッドの移動 (4)

# 4　実装

3 章で述べた手法を Java 言語を対象に実装する．本手法を実装するにあたり，STEP1 でソースコードを解析，そしてクラスの構文木を構築し，STEP3 でその構文木に修正を加えることで本手法を実現する方法が考えられる．しかし本稿では，バイトコードを対象に STEP1 のプログラム解析を行い，STEP3 では，バイトコード上で必要な修正を行い，再度バイトコードを解析することで解析情報を更新する方法で実現した．その理由としては，バイトコードはクラスファイルの構造解析・変更を行うためのライブラリが優れており，ソースコードよりも容易かつ高速に解析・変更可能なことが挙げられる．そこで，実装に用いる既存のオープンソースソフトウェアを紹介し，それらを拡張することで本手法を実装する方法を述べる．まず，本システムの構成を説明する．本システムは，大まかに以下４つのサブシステムに分けることができる．

- 入出力部
- プログラム解析部
- 解析情報修正部
- CK メトリクス計算部

次にこれらのシステム全体の概要を図 11 に示す．

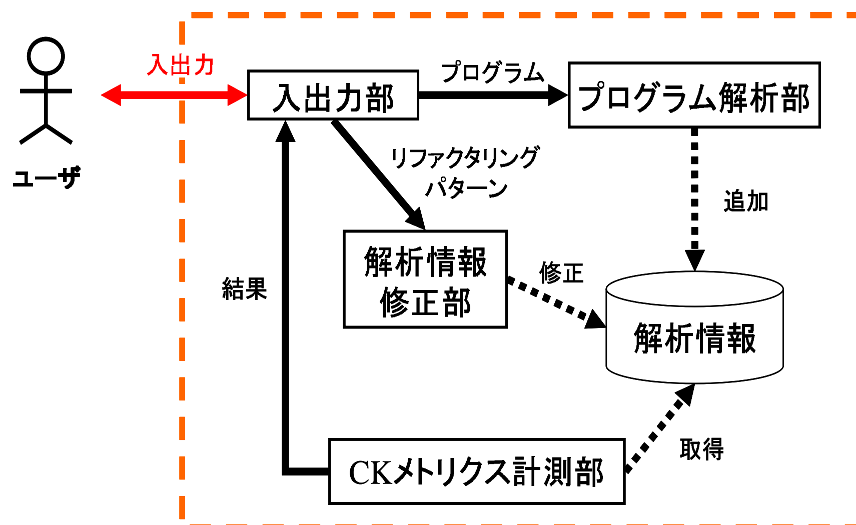



図 11: システムの概要

ユーザとシステム間の情報のやり取りは，ユーザインターフェースを備えている入出力部を介して行われる．ユーザは，入出力部でリファクタリング対象のソフトウェア，リファクタリング箇所とリファクタリングパターンの入力を行う．最初に，与えられた対象ソフトウェアは，入出力部からプログラム解析部の入力として送られる．プログラム解析部では，3.1 節で述べた解析情報を取得し，得られた解析情報をデータベースに追加する．次に，与えられたリファクタリング箇所とリファクタリングパターンは，解析情報修正部に送られる．解析情報修正部では，その情報を基に先ほどデータベースに追加した解析情報を修正する．システムはプログラム解析部と解析情報修正部の 2 つのサブシステムの処理が終わると，最後に CK メトリクス計測部が実行する．CK メトリクス計測部では，修正前後の解析情報からリファクタリング前後の CK メトリクスを計測し，それらの差分と変化率を計算する．そして，その結果を入出力部を介してユーザへ出力する．

以下では，本システムの 4 つのサブシステムについて説明していく．なお，本システムの実装に用いた環境は以下の通りである．

- CPU:Pentium4 3.20GHz
- RAM:1GB
- OS:UbuntuLinux
- JDK 1.5.0

### 4.1 入出力部

入出力部は，ユーザとシステムを仲介する部分である．具体的には，3.1 節のプログラムの入力 (図 12)，3.2 節のリファクタリング箇所とリファクタリングパターンの入力 (図 13)，3.4 節のリファクタリング前後の CK メトリクスの変化率の出力 (図 14) が行われる．

また，GUI に関しては，Class Construction Kit の GUI を基に開発した．Class Construction Kit に関しては，4.2 節で紹介する．

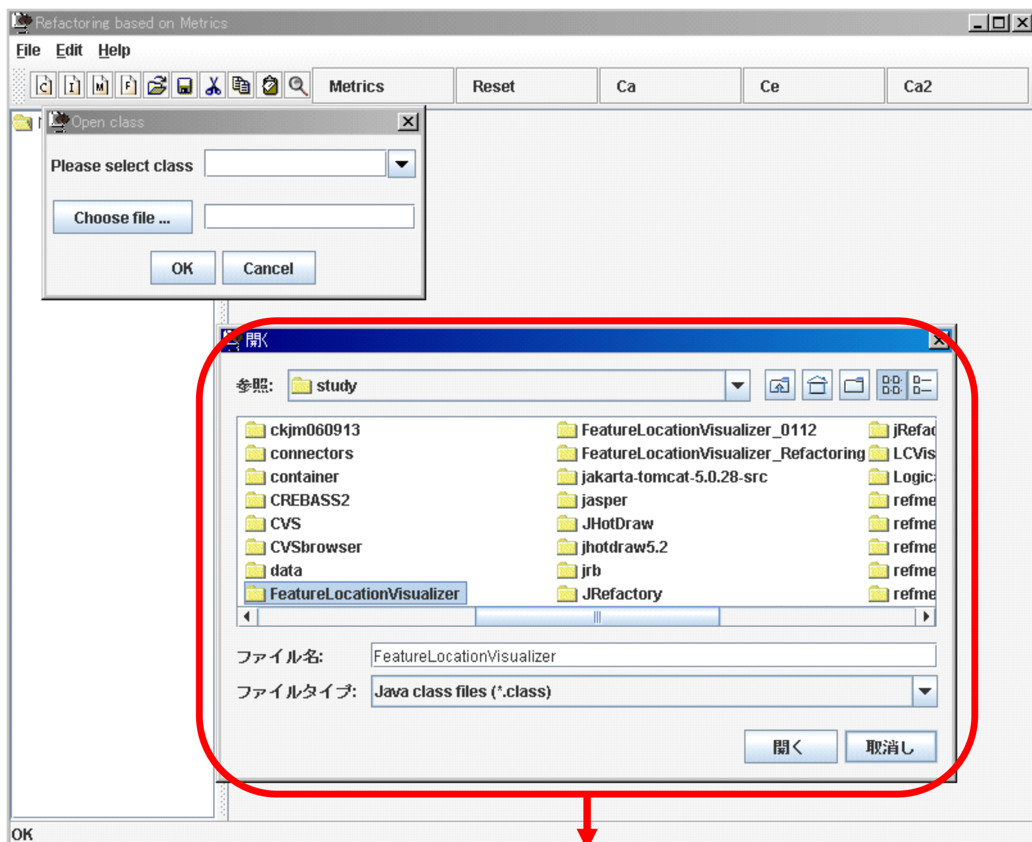



図 12: 適用対象プログラムの入力

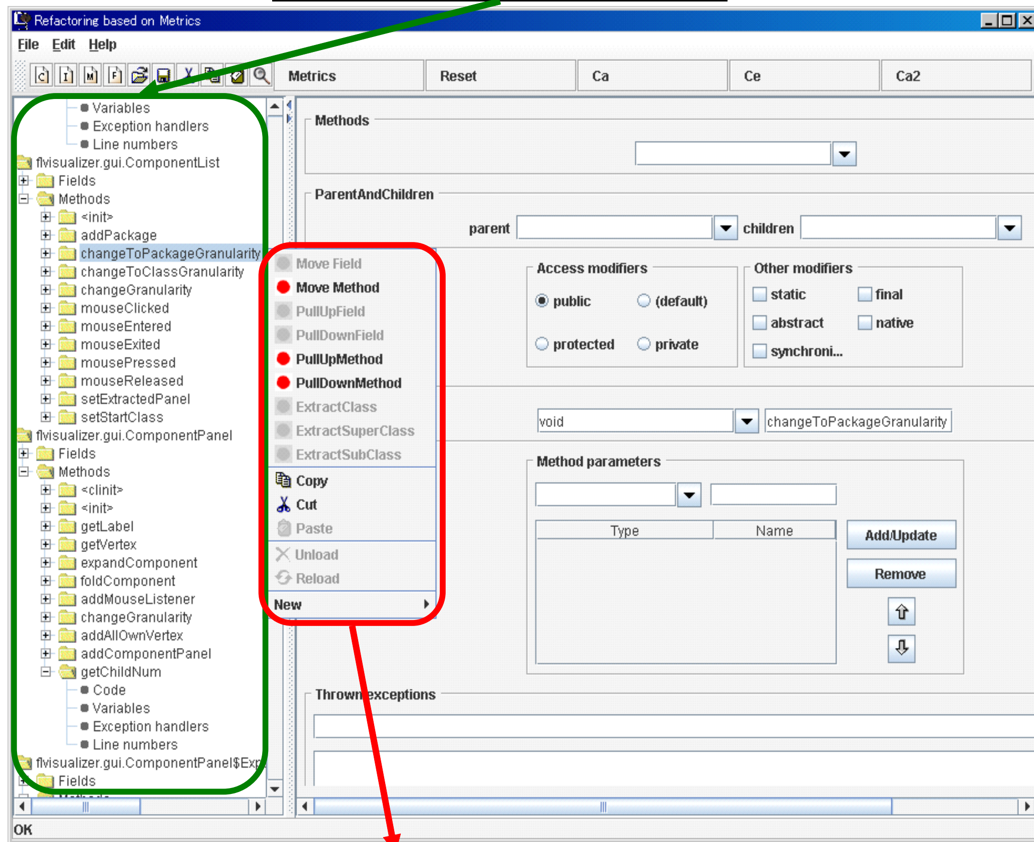



図 13: あるメソッドに対して MoveMethod の適用

CK Metrics

CKMetrics

| ClassName | WMC | DIT | NOC | CBO | Ce | Ca | RFC | LCOM | NPM |
|---|---|---|---|---|---|---|---|---|---|
| flvisualizer.FLVisualizer | 2 | 1 | 0 | 1 | 1 | 0 | 7 | 1 | 2 |
| flvisualizer.GraphVisualizer | 3 | 1 | 0 | 8 | 7 | 1 | 41 | 0 | 2 |
| flvisualizer.distance.DistanceAnalyzer | 3 | 1 | 0 | 3 | 2 | 1 | 21 | 0 | 3 |
| flvisualizer.distance.ExtendDistanceLabeler | 13 | 1 | 0 | 1 | 0 | 1 | 36 | 0 | 10 |
| flvisualizer.distance.ExtendShortestPath | 8 | 1 | 0 | 2 | 1 | 1 | 25 | 0 | 7 |
| flvisualizer.edge.callgraph.CallEdge | 4 | 2 | 0 | 3 | 1 | 2 | 10 | 4 | 4 |
| flvisualizer.edge.callgraph.CallEdgeDecorator | 3 | 1 | 0 | 0 | 0 | 0 | 7 | 0 | 3 |
| flvisualizer.edge.logicalcoupling.BundledEdge | 7 | 3 | 0 | 4 | 3 | 1 | 13 | 2 | 7 |
| flvisualizer.edge.logicalcoupling.LogicalCouplingEdge | 6 | 2 | 2 | 5 | 1 | 4 | 7 | 3 | 6 |
| flvisualizer.edge.logicalcoupling.LogicalCouplingEdgeDecorator | 3 | 1 | 0 | 2 | 0 | 2 | 7 | 0 | 3 |
| flvisualizer.extention.AlwaysFalsePickedInfo | 3 | 1 | 0 | 1 | 0 | 1 | 4 | 3 | 3 |
| flvisualizer.extention.MappedPaintFunction | 12 | 1 | 0 | 3 | 0 | 3 | 19 | 26 | 10 |
| flvisualizer.extractor.ComplementClass | 11 | 1 | 0 | 2 | 1 | 1 | 45 | 0 | 7 |
| flvisualizer.extractor.DependenceAnalyzer | 5 | 1 | 0 | 3 | 2 | 1 | 26 | 0 | 4 |
| flvisualizer.extractor.ExtractClass | 4 | 1 | 0 | 4 | 3 | 1 | 17 | 0 | 2 |

図 14: リファクタリング後の結果出力

### 4.2　プログラム解析部

プログラム解析部は，3.1 節で与えられた情報を解析する部分を実装している．入出力部で与えられたプログラムに対して解析を行い，3.1 節で述べた解析情報を取得する．プログラム解析部を実装するために Class Construction Kit というオープンソースソフトウェアを基に開発する．次に，Class Construction Kit について紹介する．

Class Construction Kit[9]

BCEL および Swing を用いて実装されているバイトコードの可視化および修正するためのツールである．ソースコードが無く，バイトコードのみからクラスの構成を知りたい場合に，このツールを使えば容易に可視化することができる．また，バイトコードを編集することでプログラムサイズの縮小が可能であったり，ソースコードレベルではできない処理を行う場合に役立つ．

次に，プログラム解析部の大まかな概要を図 15 に示す．プログラム解析部の仕組みは，まず，ユーザから入力として与えられたバイトコードを Class Construction Kit を用いて，プログラム構造を解析する．具体的には，バイトコードに含まれるすべての情報を，クラス，メソッド，フィールドといった単位で整理する．例えば,“ あるクラスに含まれるメソッド群を集合として保持し，さらにその中のメソッドが呼び出すメソッド群を集合として保持する ”といった構成となる．次に，その情報を基に 3.1 節で述べた解析情報を取得する．例えば，クラス間の参照関係を取得する場合，クラスごとにハッシュマップを準備し，すべてのクラスに対して参照するフィールドやメソッドを調べる．そして，被参照なクラスは“ 参照クラス ”を自身のハッシュマップに追加し，参照クラスも同様に被参照クラスの追加を行う．これを繰り返すことで，すべてのクラス間の参照関係が取得できる．他の解析情報を取得する際も，実装上は様々なクラスライブラリを用いて取得している．

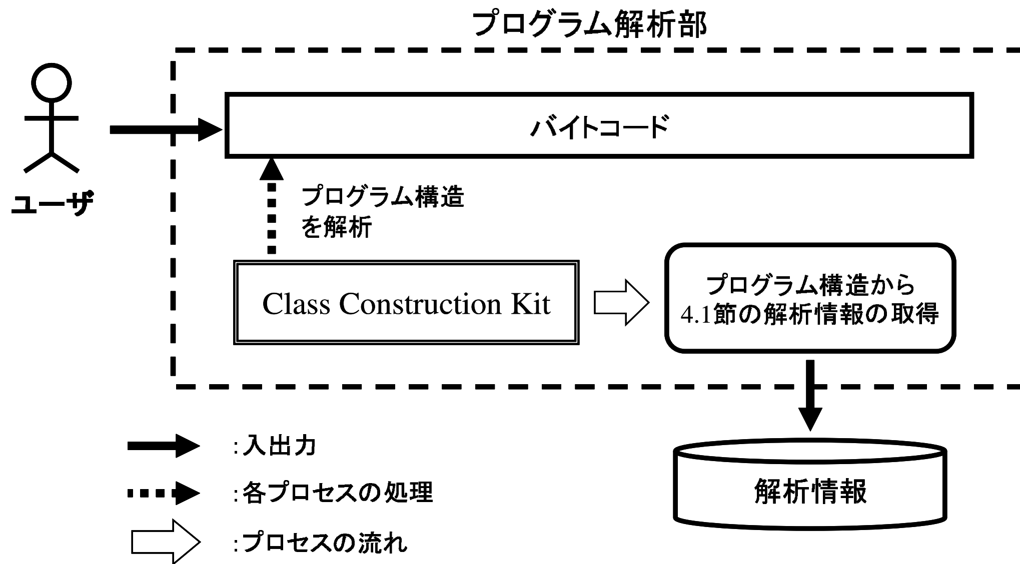



図 15: プログラム解析部の概要

### 4.3 解析情報修正部

解析情報修正部では，ユーザが与えたリファクタリング箇所とリファクタリングパターンを解析情報に反映させる．まず，解析情報修正部は，ユーザから与えられた情報を基に，リファクタリング対象となるクラスを情報を取得する．そして，その情報からリファクタリングの影響を受けるクラスを検出する．最後に，検出されたクラスの解析情報に対し，修正を行う．例えば，あるクラス A のフィールド a を他のクラスへ移動する場合，まず，フィールド a は被参照メソッド群を調べる．その被参照メソッド群が属するクラスは，すべてリファクタリングの影響を受けるクラスとして検出される．そして，被参照メソッド群のメソッド呼び出し先を，移動先のクラスに修正するという処理を行う．また，影響を受けるクラスの検出方法や解析情報の修正方法は，すべてリファクタリングパターンに依存する．

### 4.4 CK メトリクス計測部

CK メトリクス計測部では，ckjm というオープンソースソフトウェアを CK メトリクス計算機構として用いる．まず初めに，ckjm に関して説明する．

ckjm[8]

バイトコードを対象とした CK メトリクス計測ツールである．このツールは，BCEL ベースで作成されており，CK メトリクスの計測に必要な結合・継承関係の取得およ

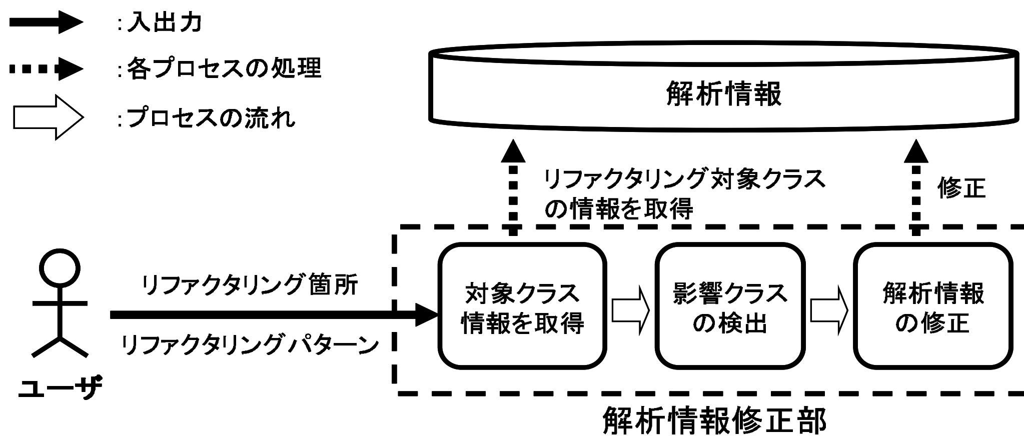



図 16: 解析情報修正部の概要

びクラスの分岐・繰り返し文の数を調べる．そして，その情報を基に CK メトリクスを計算し，いくつかの用意されたフォーマットで出力することが可能である．

ckjm は本来，プログラム解析部と CK メトリクス計測部を備えているが，本稿の実装にあたっては，CK メトリクス計測部のみを利用した．具体的には，図 17 のように，リファクタリング前後の解析情報を取得し，その情報を ckjm に与える．ckjm は，リファクタリング前後の CK メトリクスの値を出力する．そのメトリクス値から，3.4 節で述べた各メトリクスの変化率を計測し，ユーザにフィードバックする．

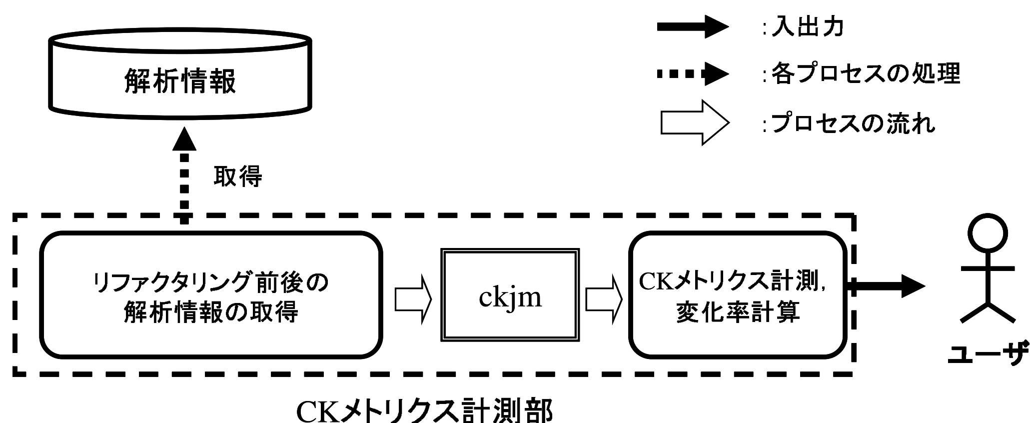



図 17: CK メトリクス計測部の概要

# 5　評価

## 5.1　実験対象プログラムの概要

本章では，4 章で実装したシステムを利用することで，いくつかのリファクタリングパターンに対し，その効果を予測する．本研究室で開発されている Feature Location Visualizer[29] を対象とした．Feature Location Visualizer の概要は表 1 の通りである．

表 1: Feature Location Visualizer の概要

| 開発期間 | 2006/5〜2007/1 |
|---|---|
| ソースファイル数 | 37 |
| クラス数 | 37 |
| 行数 | 4815 |

## 5.2　実験概要

Feature Location Visualizer の機能設計が不十分で改善すべき箇所を開発者と共に調査を行った．次に，開発者はその箇所に対して適切だと判断したリファクタリングによる改善案をいくつか挙げる．実際の問題点と改善案を以下に示す.

**問題点**：

本来 GUI とは関係のない ComponentList（以下 CL）クラス内に，GUI 関連の処理が実装されている．具体的には，CL クラス内に ExtractPanel（以下 EP）クラスがローカル変数として宣言され，その変数を参照するメソッド setExtractPanel()，setStartClass() が存在する．

**改善案**：

その変数とメソッド群を GUI の処理を行うクラスに移動することを考える，具体的には，GUI の機能を実装している以下に示すクラスが移動先の候補である．また，各移動先クラスに移動するリファクタリングをそれぞれ case1〜4 と呼称する．

**case1** ：FeatureLocationGraphVisualizationViewer（FLGVV）クラス

**case2** ：BirdPanel（BP）クラス

**case3** ：ComponentPanel（CP）クラス

**case4** ：GraphViewPanel（GVP）クラス

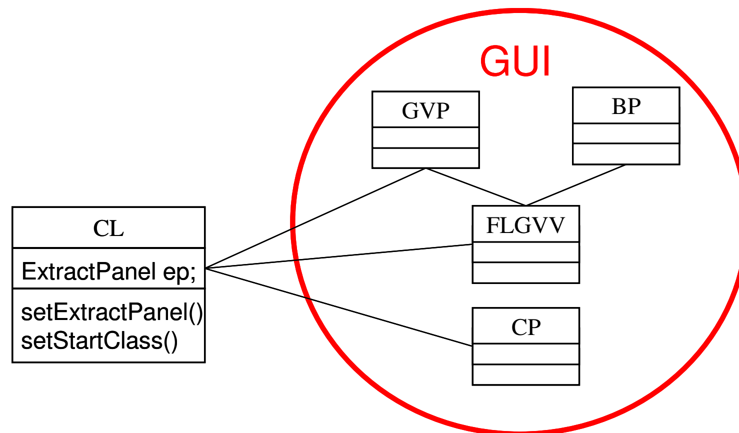



図 18: Feature Location Visualizer の問題点

この 4 通りのリファクタリングに関して，本手法を用いてリファクタリングの効果を予測する実験を行い，結果に対して比較および考察を行う．

### 5.3 実験結果

初めに，この 4 つの case に関して影響を受けるクラス群のメトリクス (リファクタリング前) を表 2 に示した．また，case1〜4 に対してそれぞれ実験を行い，3 章で提案した手法の STEP4 で得られたリファクタリング前後の CK メトリクスと，その差分と変化率を計測した．表 3〜6 は，各クラスのリファクタリング後のメトリクス値とリファクタリング前後のメトリクスの増減を括弧内に記載している．さらに，各メトリクスの合計と各メトリクスの合計の変化率も併せて記載した．各表には，リファクタリング前後で CK メトリクスの変化があったクラスのみを記載している（解析情報を修正したクラスが必ずしもメトリクスが変化するとは限らない）．また，case ごとで CK メトリクスが変化したクラスの数に違いが表れていることがわかる．case1 は ComponentList クラス, FeatureLocationGraphVisualizationViewer クラス, ExtractPanel クラスの 3 クラス，同様に case2 は 5 クラス，case3 は 3 クラス，case4 は 4 クラスである．適用したリファクタリングパターンは各 case で共通であり，単に移動先のクラスを変えただけだが，構造的な変化の影響範囲は case で差が表れることがわかる．次に，これらの結果について，考察を行う．

表 2: リファクタリングの影響を受けるクラス群のメトリクス値

| クラス名 | WMC | DIT | NOC | CBO | RFC | LCOM |
|---|---|---|---|---|---|---|
| BirdPanel(BP) | 4 | 1 | 0 | 6 | 30 | 0 |
| ComponentList(CL) | 12 | 1 | 0 | 9 | 38 | 62 |
| ExtractPanel(EP) | 5 | 1 | 0 | 14 | 83 | 0 |
| GraphViewPanel(GVP) | 6 | 1 | 0 | 6 | 36 | 0 |
| ComponentVertex(CV) | 16 | 1 | 2 | 13 | 24 | 50 |
| FeatureLocationGraphVisualizationViewer(FLGVV) | 19 | 1 | 0 | 12 | 68 | 1 |
| ComponentPanel(CP) | 11 | 1 | 2 | 7 | 39 | 1 |

表 3: case1 の結果

| クラス名 | WMC | DIT | NOC | CBO | RFC | LCOM |
|---|---|---|---|---|---|---|
| ComponentList | 10(-2) | 1 | 0 | 8(-1) | 36(-2) | 62 |
| FeatureLocationGraphVisualizationViewer | 21(+2) | 1 | 0 | 12 | 71(+3) | 1 |
| ExtractPanel | 5 | 1 | 0 | 13(-1) | 83 | 0 |
| 合計 | 36 | 3 | 0 | 33(-2) | 190(+1) | 63 |
| 変化率 (%) | 0.00 | 0.00 | 0.00 | -5.71 | 0.53 | 0.00 |

表 4: case2 の結果

| クラス名 | WMC | DIT | NOC | CBO | RFC | LCOM |
|---|---|---|---|---|---|---|
| ComponentList | 10(-2) | 1 | 0 | 9 | 36(-2) | 62 |
| BirdPanel | 6(+2) | 1 | 0 | 9(+3) | 33(+3) | 0 |
| ExtractPanel | 5 | 1 | 0 | 13(-1) | 83 | 0 |
| GraphViewPanel | 6 | 1 | 0 | 7(+1) | 36 | 0 |
| ComponentVertex | 16 | 1 | 2 | 14(+1) | 24 | 50 |
| 合計 | 43 | 5 | 2 | 52(+4) | 212(+1) | 112 |
| 変化率 (%) | 0.00 | 0.00 | 0.00 | 8.33 | 0.47 | 0.00 |

表 5: case3 の結果

| クラス名 | WMC | DIT | NOC | CBO | RFC | LCOM |
|---|---|---|---|---|---|---|
| ComponentList | 10(-2) | 1 | 0 | 8(-1) | 36(-2) | 62 |
| ComponentPanel | 13(+2) | 1 | 2 | 9(+2) | 42(+3) | 1 |
| GraphViewPanel | 6 | 1 | 0 | 7(+1) | 36 | 0 |
| 合計 | 29 | 3 | 2 | 24(+2) | 84(+1) | 63 |
| 変化率 (%) | 0.00 | 0.00 | 0.00 | 9.09 | 1.20 | 0.00 |

表 6: case4 の結果

| クラス名 | WMC | DIT | NOC | CBO | RFC | LCOM |
|---|---|---|---|---|---|---|
| ComponentList | 10(-2) | 1 | 0 | 9 | 36(-2) | 62 |
| GraphViewPanel | 8(+2) | 1 | 0 | 7(+1) | 38(+2) | 0 |
| ExtractPanel | 5 | 1 | 0 | 13(-1) | 83 | 0 |
| ComponentVertex | 16 | 1 | 2 | 14(+1) | 24 | 50 |
| 合計 | 39 | 4 | 2 | 43(+1) | 181 | 112 |
| 変化率 (%) | 0.00 | 0.00 | 0.00 | 2.38 | 0.00 | 0.00 |

### 5.4　考察

前節の実験結果の中で，各メトリクスの変化に関して最初に考察を行う．まず，継承に関する複雑さを計測する NOC，DIT は，すべての case において全く変化が表れなかった．この理由としては，今回の適用した“ フィールドの移動 ”,“ メソッドの移動 ”といったリファクタリングパターンが，継承関係の変化を伴わないからである．次に，クラスの内部複雑度を計測する WMC,LCOM について考察する．WMC は，計測対象のクラスのメソッドの複雑さを合計することで計測する．すべての case において，移動元クラスと移動先クラスで同じ値だけ WMC の値が増減していることがわかる．これは“ メソッドの移動 ”によって，メソッドの複雑度が移動元クラスと移動先のクラスの間で移転したことを表している．LCOM は，getter/setter は省いて計測する仕様を採用したため，値に変化が見られなかった．最後に，結合の複雑さを計測する CBO,RFC について考察する．クラス間の結合関係の複雑さを表す CBO は，移動先のクラスで変わらない (case1)，もしくは，増加している (case2,3,4) ことがわかる．移動先のクラスで CBO が変わらない場合，移動先のクラスは，移動したフィールドのクラス型や移動したメソッドの呼び出し先のクラスと既に結合関係がある，一方，CBO が増加した場合は移動先クラスは結合関係がなかったことを意味する．RFC は，case ごとに移動先のクラスで異なる．リファクタリング前に移動先のクラスで“ 移動するメソッド ”を参照していた場合，移動後は移動先のクラス自身に参照先に変更される．一方，参照していない場合は移動先でそのような参照の変更は発生しない．それに伴って移動先のクラスが RFC が異なると考えられる．

次に，各 case の変化率をクラス数で正規化し，比較を行った（表 7）．表 7 の結果から，“ フィールド，メソッド (setter) の移動 ”では，すべての case において WMC の合計値は変化がなく，RFC の合計値は，微量な変化率を示すことがわかった．また，図 19 から各 case の比較を行ったところ，case1 はリファクタリング後の CBO の値が大きく減少し，最も改善していることがわかる．case2,3,4 はリファクタリング後の CBO の値が増加し，構造的には悪化してしまうという結果が得られた．

本実験では，4 つのクラスを移動先として挙げたが，これらはすべて開発者が意味的な不整合がないかを考慮し，自発的に挙げたものである．ここで，開発者に対し，この 4 つの case で良いと思われる case を主観で選択してもらう調査を行った．

開発者は，移動するフィールドのクラス型が，case3 の移動先クラスの型と機能の内容が似ているという理由から case3 を選択した．しかし，今回の実験結果では，case3 は他の case と比較したところ結合に対する複雑さが大きく増加し，不適切であるという結果だった．この結果を開発者に考慮してもらい，もう一度ヒアリングしたところ，case1 を採用することになった．これらのことから，開発者はソースコード上からソフトウェアの複雑さを把握し，

それを考慮してリファクタリングを行うのは容易でないことがわかる．本手法は，そのような開発者が容易に把握できないソフトウェアの複雑さの理解を助けることが可能であり，有用であると言える．

また，本手法では，リファクタリング前後のメトリクスの値だけでリファクタリングを評価している．しかし，実際のリファクタリングは，ソースコード修正やテストにもコストがかかるため，より正確に評価するためにはこれらのコストも考慮するべきであると考えている．

表 7: 各 case の変化率をクラス数で正規化

| | WMC | DIT | NOC | CBO | RFC | LCOM |
|---|---|---|---|---|---|---|
| case1 | 0.00 | 0.00 | 0.00 | -1.90 | 0.18 | 0.00 |
| case2 | 0.00 | 0.00 | 0.00 | 1.67 | 0.09 | 0.00 |
| case3 | 0.00 | 0.00 | 0.00 | 3.03 | 0.40 | 0.00 |
| case4 | 0.00 | 0.00 | 0.00 | 0.60 | 0.00 | 0.00 |

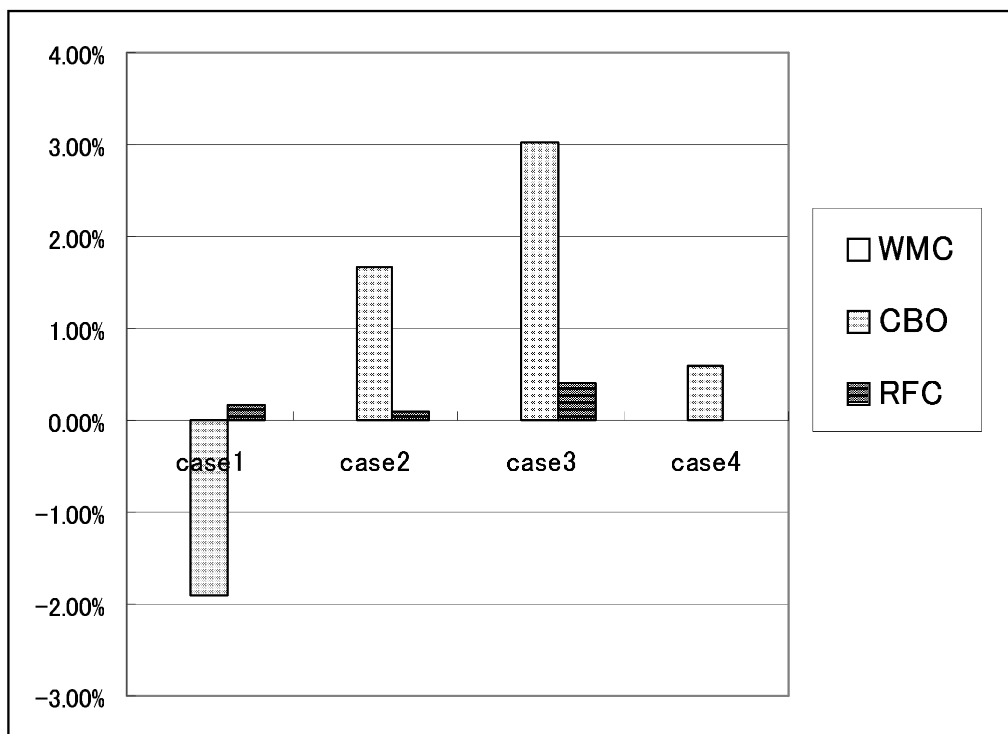



図 19: 各 case の比較

# 6　関連研究

## 6.1　リファクタリングの効果計測手法

Kataoka らはリファクタリングがソフトウェア保守に与える効果を定量的に計測する手法を提案している [2]．具体的には，高結合を解消する効果的なリファクタリングに対し，独自に定義した結合度メトリクスを用いて，ソフトウェアの保守性に与える効果を計測するというものである．この研究では，結合度というソフトウェア品質や保守の容易さを決定する重要な要因の 1 つで評価を行っているが，リファクタリングは結合度以外の様々の属性に影響を与える．そのため，その評価と保守性に与える効果の合理性に関する考察が必要であると考える．

## 6.2　リファクタリングの適用支援ツール

リファクタリングは，ソースコードの修正コストを要する．特にソースコードに対して多くの前提条件件を検査すること， および，変換後のソースコードが影響を及ぼす箇所を特定することが非常に面倒である．そこで，リファクタリングの修正コストを軽減するために，リファクタリング適用支援に関する研究が行われてきている [22][16][17][24][11]．これらの研究で提案している支援ツールを使用すると，“ メソッド名の変換 ”など，構造的な変化を伴わないリファクタリングは，自動的に変換が可能である．しかし，構造的な変化を伴うリファクタリング，つまり“ メソッドの移動 ”や“ クラスの抽出 ”といったクラスの結合や継承に変化を与えるリファクタリングはうまく適用できない場合がある．例えば，移動しようとするメソッドが他のクラスのメソッドから参照されており，影響するクラスを修正するのに，開発者に依存するような変換に対しては，ソースコードを手動で修正する必要がある．このように，支援ツールを用いたとしても，ソースコードの自動修正には様々な課題があり，現段階では開発者は修正コストを必要とする．これらのことから，実際にソースコードを修正する前にリファクタリングが保守性に与える効果を評価することは，適切なリファクタリングを行うための合理的な方法であると考える．

## 7　まとめ

本稿では，ソースコードを修正する前にリファクタリングがソフトウェア保守に与える効果を予測する手法を提案・実装し，そのツールの有用性を評価した．具体的には，ソフトウェア保守の容易さを評価するために用いられるCKメトリクスに用いて，リファクタリングが保守性に与える効果を予測する手法を提案した．さらに，本手法を実装するために，2つのオープンソースソフトウェアを紹介し，それらを基に開発したシステムの構成を述べた．また，本システムの有用性を確認するために，研究室で開発中のツールを対象とし4つのケースに対して適用実験を行った．適用実験で得られた4つのケースのリファクタリングの効果予測結果に対する考察と，本システムの有用性の評価を行った．また，今後の課題としては，以下のことがあげられる．

- “ クラスのインライン化 ”など，構造的な変化を伴う未実装なリファクタリングに関しても実装する
- ソースコード修正にかかるコスト，テストにかかるコストを考慮し，手法を改良する
- 実装している他のリファクタリング（クラスの抽出など）に関して，評価を行う．

# 謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました 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 井上 克郎 教授に心より深く感謝致します.

本論文を作成するにあたり，常に適切な御指導，御助言を賜りました 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 楠本 真二 教授に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を賜りました 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 松下 誠 助教授に心から感謝致します.

本研究において，様々な御協力を頂きました 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 肥後 芳樹 氏に深く感謝します.

最後に，その他様々な御指導，御助言等を頂いた 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 井上研究室の皆様に深く感謝いたします.

## 参考文献

[1] *IEEE Std 1219: Standard for Software Maintenance*. 1997.

[2] A Quantitative Evaluation of Maintainability Enhancement by Refactoring. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, p. 576, Washington, DC, USA, 2002. IEEE Computer Society.

[3] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Softw. Eng.*, Vol. 22, No. 10, pp. 751–761, 1996.

[4] Bart Du Bois, Serge Demeyer, and Jan Verelst. Refactoring - Improving Coupling and Cohesion of Existing Code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pp. 144–151, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw.*, Vol. 51, No. 3, pp. 245–273, 2000.

[6] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*, Vol. 20, No. 6, pp. 476–493, 1994.

[7] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer. Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Trans. Softw. Eng.*, Vol. 24, No. 8, pp. 629–639, 1998.

[8] ckjm. `http://www.spinellis.gr/sw/ckjm/`.

[9] Class Construction Kit. `http://bcel.sourceforge.net/cck.html`.

[10] M. Dorfman and R. H. Thayer. *Software Engineering*. IEEE Computer Society Press, 1997.

[11] Eclipse. `http://www.eclipse.org/`.

[12] N. Ford and M. Woodroffe. *Introducing software engineering*. Prentice-Hall, 1994.

[13] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[14] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[15] V. Porter L. Brian J. Daly and J. Wüst. Predicting Fault-Prone Classes with Design Measures in Object-Oriented Systems. In *ISSRE '98: Proceedings of the The Ninth International Symposium on Software Reliability Engineering*, p. 334, Washington, DC, USA, 1998. IEEE Computer Society.

[16] jFactory. `http://www.instantiations.com/jfactor/`.

[17] JRefactory. `http://jrefactory.sourceforge.net/`.

[18] Pigoski T. M. *Maintenance*. Encyclopedia of Software Engineering, 1994.

[19] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, p. 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[20] Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.*, Vol. 30, No. 2, pp. 126–139, 2004.

[21] Metamata Metrics. `http://www.metamata.com/`.

[22] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theor. Pract. Object Syst.*, Vol. 3, No. 4, pp. 253–263, 1997.

[23] Tom Tourwé; and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, p. 91, Washington, DC, USA, 2003. IEEE Computer Society.

[24] Transmogrify. `http://transmogrify.sourceforge.net/`.

[25] E. J. Weyuker. Evaluating Software Complexity Measures. *IEEE Trans. Softw. Eng.*, Vol. 14, No. 9, pp. 1357–1365, 1988.

[26] S. W. L. Yip and T. Lam. A software maintenance survey. Proc. of APSEC '94, pp. 70–79, 1994.

[27] 秦野克彦, 乃村能成, 谷口秀夫, 牛島和夫. ソフトウェアメトリクスを利用したリファクタリングの自動化支援機構. 情報処理学会論文誌, 第 44 巻, pp. 1548–1557, 2003.

[28] 神谷年洋. オブジェクト指向メトリクスを用いた開発支援法に関する研究. PhD thesis, 大阪大学大学院基礎工学研究科, 2001.

[29] 檜皮祐希, 松下誠, 井上克郎. 更新履歴情報と静的情報を用いて同一機能を実装しているクラス群を抽出する手法の提案. 信学技報, 第 106 巻, pp. 13–18, 2006.

# 付録

A. リファクタリングパターン

A.1 フィールドの移動

A.2 フィールドの引き上げ

A.3 フィールドの引き下げ

A.4 メソッドの移動

A.5 メソッドの引き上げ

A.6 メソッドの引き下げ

A.7 クラスの抽出

A.8 スーパークラスの抽出

A.9 サブクラスの抽出

# A　リファクタリングパターン

ここでは，本稿で実装したリファクタリングパターンについて紹介する．

## A.1　フィールドの移動

あるクラスに定義されているフィールドが，現在または将来に渡って，定義しているクラスよりも他のクラスから使われることの方が多い場合がある．その際，多く使っているクラスにフィールドを移動し，そのフィールドの利用側をすべて変更する (図 20 参照).

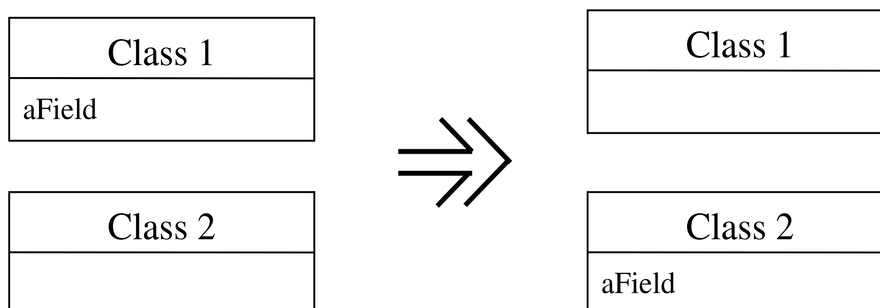



図 20: フィールドの移動

## A.2　フィールドの引き上げ

2 つのサブクラスが同じフィールドを持っている場合に，そのフィールドをスーパークラスに移動する (図 21 参照).

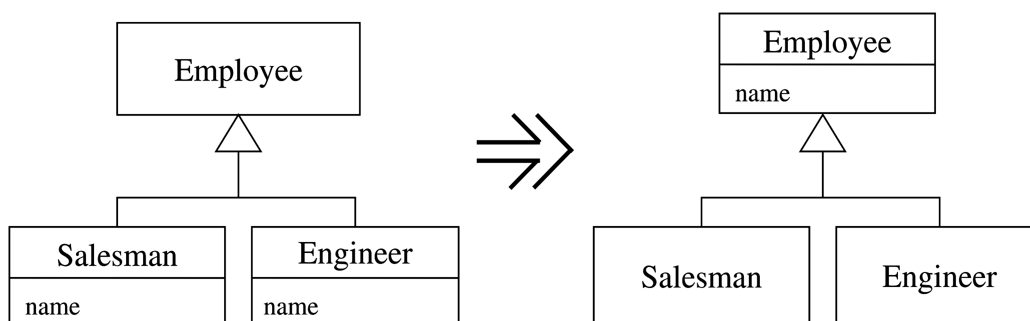



図 21: フィールドの引き上げ

### A.3　フィールドの引き下げ

フィールドが幾つかのサブクラスだけに使われている場合に，そのフィールドをそれらのサブクラスに移動する (図 22 参照).

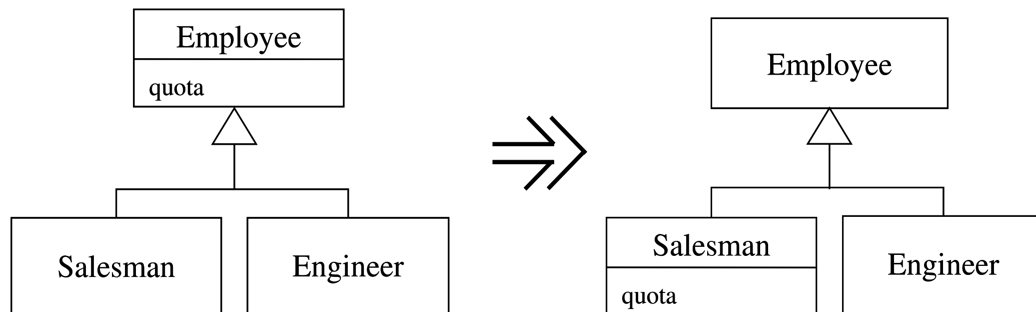



図 22: フィールドの引き下げ

### A.4　メソッドの移動

あるクラスでメソッドが定義されているが，現在または将来に渡って，そのクラスの特性よりも他のクラスの特性の方がそのメソッドを使うか，そのメソッドに使われることが多い場合がある．その際，同様の本体を持つ新たなメソッド，最も多用するクラスに作成する．元のメソッドは単純な委譲とするか，または取り除く (図 23 参照).

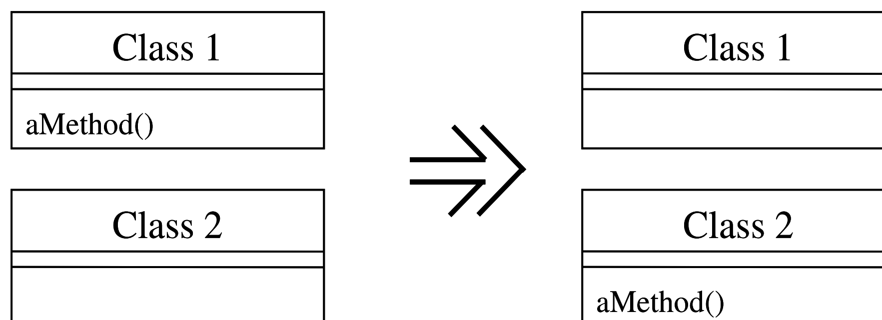



図 23: メソッドの移動

### A.5　メソッドの引き上げ

同じ結果をもたらすメソッドが複数のサブクラスに存在した場合，それらを親クラスに引き上げる．最も単純なケースは，複数のメソッド本体が全く同じである場合である (図 24 参

照). 重複したコードが兄弟クラスに存在した場合には，メソッドの抽出を行ってから，メソッドの引き上げを行えばよい (図 24 参照).

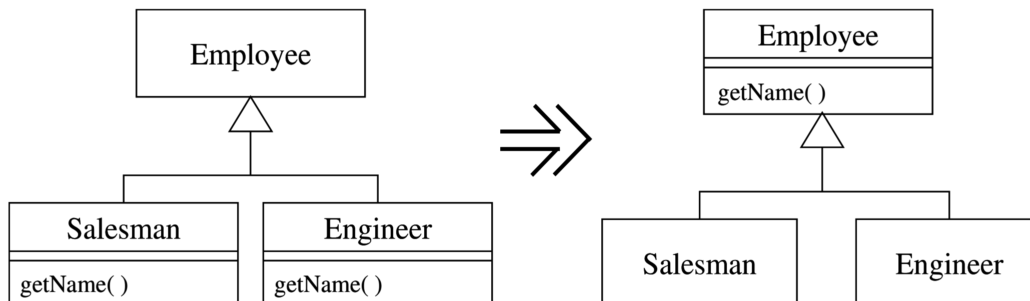



図 24: メソッドの引き上げ

### A.6　メソッドの引き下げ

スーパークラスの振る舞いが，幾つかのサブクラスだけに関係している場合に，そのメソッドをサブクラスに移動する (図 25 参照).

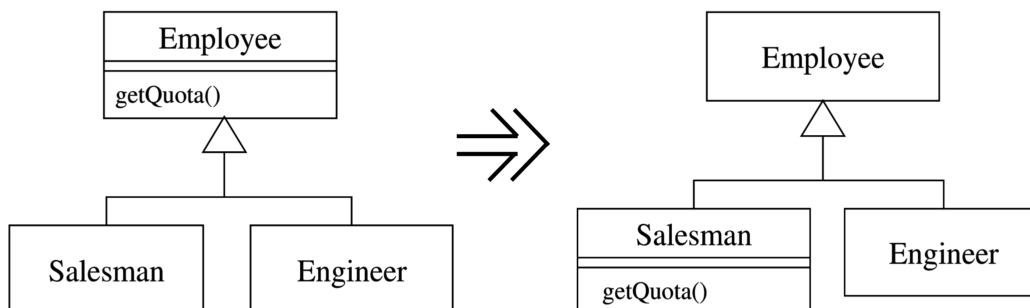



図 25: メソッドの引き下げ

### A.7　クラスの抽出

2 つのクラスでなされるべき作業を 1 つのクラスで行っている際に，新たにクラスを作って，適当なフィールドとメソッドを元のクラスからそこに移動する (図 26 参照).

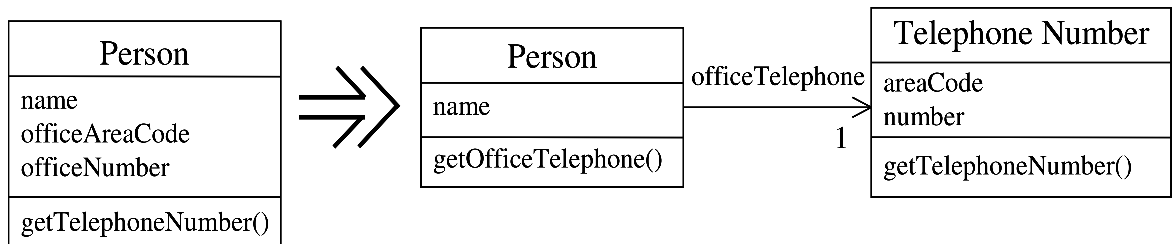



図 26: クラスの抽出

### A.8 スーパークラスの抽出

似通った特性を持つ複数のクラスがある場合に，新たに親クラスを作成して，共通の特性を移動する．クラスの抽出との違いは，継承するか委譲するかの違いである (図 27 参照).

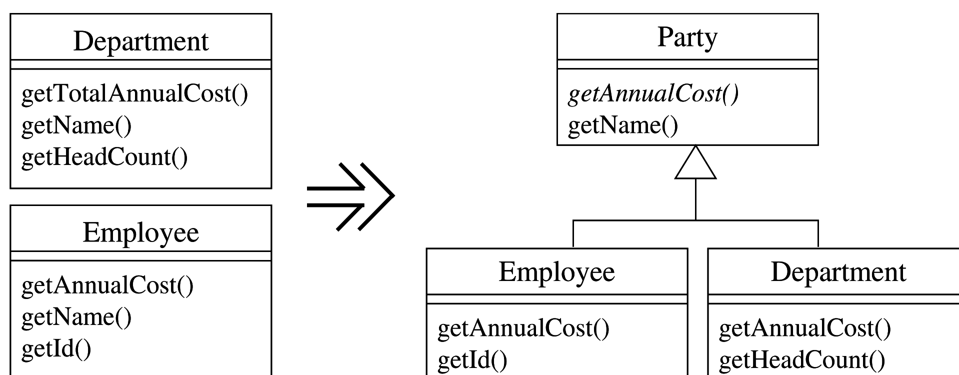



図 27: スーパークラスの抽出

### A.9 サブクラスの抽出

あるクラスの特定のインスタンスだけに必要な特性がある場合に，新たに子クラスを作成して，その一部の特性を移動する．クラスの抽出との違いは，継承するか委譲するかの違いである (図 28 参照).

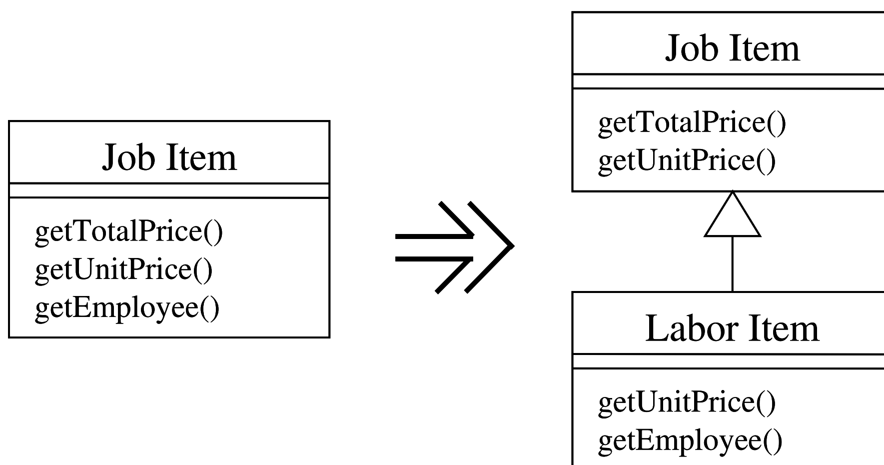



図 28: サブクラスの抽出