社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE

CK メトリクスに基づくリファクタリングの効果予測手法の提案と実装

松本　義弘† 　肥後　芳樹† 　楠本　真二† 　井上　克郎†

†大阪大学 大学院情報科学研究科 〒560-8531 大阪府豊中市待兼山町 1-3
E-mail: †{y-matsu,y-higo,kusumoto,inoue}@ist.osaka-u.ac.jp

あらまし　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，そのツール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 メトリクスに基づき，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，本手法の適用実験と評価を行い，本手法の有用性を示した．

キーワード　リファクタリング，CK メトリクス，ソフトウェア保守

Approach and implementation of refactoring effects prediction using CK Metrics

Yoshihiro MATSUMOTO†, Yoshiki HIGO†, Shinji KUSUMOTO†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1-3. Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
E-mail: †{y-matsu,y-higo,kusumoto,inoue}@ist.osaka-u.ac.jp

Abstract　Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. There are several previous researches that aim to detect candidates for refactoring and of choosing the appropriate refactoring pattern. Especially, these methods support refactorings for structural fault (high cohesion, low coupling). On the other hand, developers refactor programs from the viewpoints of not only structual but also functional complexity. Therefore, developers often have to evaluate the refactoring effect. In this paper, we propose a prediction method of refactoring effects to the software maintainability using CK Metrics. We applied the proposed method to a certain program and showed the usefulness and applicability of it.

Keyword　refactoring, CK metrics, software maintenance

1. はじめに

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[1]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][5][6]．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．

以降，2 章では，関連研究について紹介し，3 章では，ソフトウェア保守と CK メトリクスについて述べる．4 章では，提案する手法について述べ，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う．6 章では，適用実験の結果に関する考察および本ツールの有用性の評価を行い，最後に，7 章でまとめと今後の課題を述べる．

2. 関連研究

2.1. 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[3]．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．

2.2. 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[7][8][9][10][15]．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．

3. 準備

3.1. ソフトウェアの保守性

ソフトウェアの品質や保守の容易さを評価／予測するためにソフトウェア複雑度メトリクスが用いられる[13]．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エラーが含まれている可能性が高く(品質が低く)，保守が困難であると評価されている．また，代表的なソフトウェア複雑度メトリクスには，ChidamberとKemererが提案したCKメトリクスがある[2]．

3.2. CK メトリクス

CKメトリクスは，ChidamberとKemererによって開発されたオブジェクト指向設計を対象とする 6 つのメトリクスであり，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[12]をほぼ満たすことが確認されている[2]．また，BasiliらはCKメトリクスを実験的に評価し，従来のコードメトリクスの組み合わせよりも，フォールトの発生を予測するための良い指標となることを示した[11]．CKメトリクスはクラスを対象としてその複雑さを評価するものであり，以下のように定義されている．

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，$WMC(C)=\Sigma c_i$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[16]，McCabeのサイクロマチック数[17] などを用いる方法が考えられる．本稿ではサイクロマチック数を用いることとする．

DIT(Depth of Inheritance Tree):

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)からそれ以上基底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):

RFC は，ローカルメソッドの数とリモートメソッドの数の和で定義される．ローカルメソッドは計測対象のクラスに定義されているメソッド，リモートメソッドはローカルメソッドから呼び出される，かつ計測対象のクラス自身で定義されていないメソッドとして定義される．

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．$I_i(i=1,...,n)$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$\phi$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．

4. 提案手法

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，以下 4 つの STEP を通じて，CK メトリクスを計測する．

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：CK メトリクスの再計算

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が必要とする．次に，この手法に関して詳しく説明する．

4.1. 対象プログラムの解析および CK メトリクス計測（STEP1）

CKメトリクスを計算するために対象プログラムを解析する．解析する情報は，すべてのクラスに対するフィールド，メソッドの情報および，CKメトリクスを計算するために必要な結合・継承関係，クラス内のフィールドとメソッドの参照関係，メソッド間の参照関係が含まれる．CBO,RFC,NOC,DITは，すべてのクラスの結合や継承関係から計算される．WMCは，メソッドの複雑度にMcCabeのサイクロマチック数[17]を用いるため，メソッド内の分岐や繰り返し文の構成を元に計算，LCOMはフィールド・メソッドの関係から計算する．

4.2. 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）

リファクタリングしたい箇所（フィールド，メソッド，クラス）を指定し，適用したい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，その中のいくつかを実装した．現在のところ，対称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（引き上げ・引き下げ）
- メソッドの移動（引き上げ・引き下げ），
- クラスの抽出（スーパークラス，サブクラスの抽出を含む）

なお，実装に関しては 5 章で詳しく述べる．

4.3. 解析情報の修正（STEP3）

4.1 節で取得した解析情報と 4.2 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，リファクタリングの影響がおよぶクラス群をすべて特定する．そして，そのクラス群に対する解析情報の修正を行う．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」を適用する例を挙げ，具体的に解析情報の修正方法について説明する．クラスA，B と，メソッドa1 の呼び出し関係を図 1(a)のような関係であると仮定する．また，解析情報の修正は，「自動修正」と「ユーザ依存箇所の修正」に分けられる．

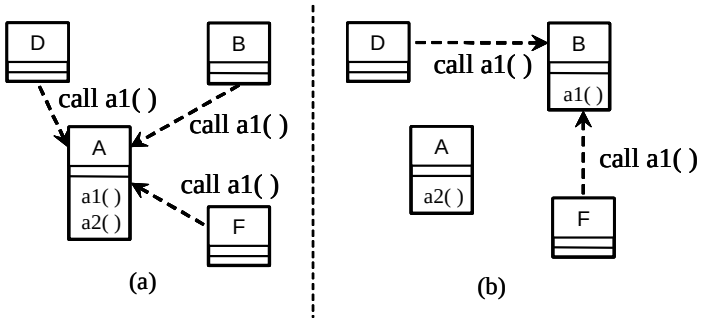

図 1. クラス A から B へのメソッドの移動

4.3.1. 自動修正

自動修正は，ユーザに依存せず機械的に修正できるものに限られる．例えば，「クラス A からメソッド a1 を削除し，クラス B にメソッド a1 を追加する」，「クラス B から A のメソッド a1 への参照を B の内部メソッド参照に変更する」が挙げられる．ユーザに依存する処理に関しては，次項で説明する．

4.3.2. ユーザ依存箇所の修正

「メソッドの移動」のような構造的な変化を伴うリファクタリングは，機械的に修正できないユーザ依存箇所がある．例えば，図 1(b)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 を参照するのにどのインスタンスから呼び出すかを決定する必要がある．本手法では，解析情報からこのユーザ依存の修正箇所を自動検出し，その箇所に関してユーザから回答を求める．そして，その回答を元に修正を行う．

4.4. CK メトリクスの再計算（STEP4）

4.1 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，その計測結果と 4.1 節で計測した CK メトリクスの差分を計算し，その結果を示す．

5. 適用実験

5.1. 実装

本手法の有用性を確認するために，4 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．

5.2. 実験概要

本研究室で開発されている”Feature Location Visualizer[14]”（以下，FLV）を対象に行った適用実験に関して述べる．FLVはクラス数 37 クラス，行数 4815 行である．まず，FLVに対し，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する．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 setExtractPanel メソッド，setStartClass メソッドが存在する．

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，FeatureLocationGraphVisualizationViewer（以下FLGVV）クラス，ComponentPanel（以下CP)クラスのいずれかへ移動する．その変数とメソッド群をFLGVVクラスへ移動するリファクタリングをパターン１，CPへ移動するリファクタリングをパターン２とする（図 2）．

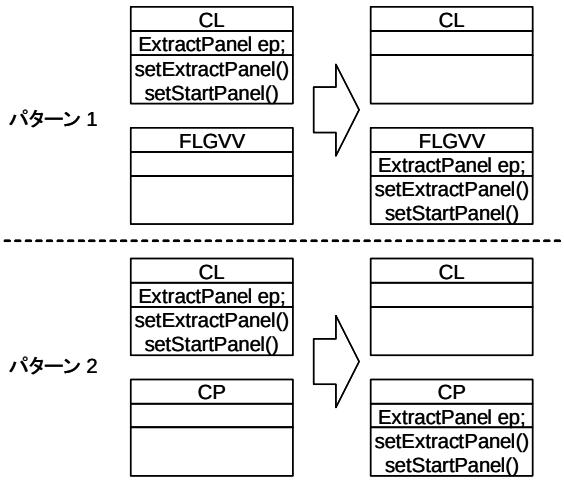

図 2. 改善案（パターン 1,2）

この 2 通りのリファクタリングに関して，ツールを用いて評価を行う．4 章で提案した計測手順の STEP4 で得られたリファクタリング前後の CK メトリクスの差分の結果を表 1 と表 2 に示す．

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を記載している．また，今回のリファクタリングは，継承関係の変化を伴わないので，NOC，DIT は省略する．

表 1. パターン 1 の結果

クラス名	WMC	CBO	RFC	LCOM
CL	−2	−1	−2	−19
FLGVV	2	0	3	37
EP	0	−1	0	0

表 2. パターン 2 の結果

クラス名	WMC	CBO	RFC	LCOM
CL	−2	−1	−2	−19
CP	2	1	3	21

5.3. 結果（評価）

影響の及ぶ範囲のクラス群のリファクタリング前後の CK メトリクスを得ることで，開発者にとってどのような有益があるかを検証する．

まず始めに，パターン 1 とパターン 2 で同じ結果が得られた WMC と RFC について述べる．WMC は，setExtractPanel，setStartClass メソッドの WMC が合計 2 であり，移動元と移動先で同じ変化量である．RFC は，移動元のクラスと移動先のクラスで，ローカルメソッドとリモートメソッドの数の変化量に違いが現れる．

LCOM は，パターン 1 とパターン 2 で，移動元クラスの変化量よりも，移動先クラスの変化量の方が大きい（悪化している）．しかし，本適用実験で移動するよ

うなフィールドへの setter（setExtractPanel，setStartClass）を含む場合，一概に悪化したとは言えない．なぜなら，他のすべてのメソッドはその setter を通じて変数アクセスを行うため，すべて，間接的にはフィールドアクセスとなる．LCOM の定義上，各メソッドのフィールドの共有アクセスは，直接的なアクセスしか考慮しておらず，移動したフィールドを直接アクセスするのはその setter のみとなり，LCOM の値は大きく増加してしまう（また，その値の幅はメソッドの数に依存する）．

CBOは，パターン 1 とパターン 2 で異なる結果が得られた．パターン 1，2 共に，変数とメソッド群を移動することでCLクラスとEPクラスの結合関係はなくなる．ただし，パターン 2 では移動先のCPクラスで新たにEPクラスとの結合関係が新たに発生する一方で，パターン 1 では，既にFLGVVクラスとEPクラスとの結合関係があるため，新たな結合は発生しない（図 3）．

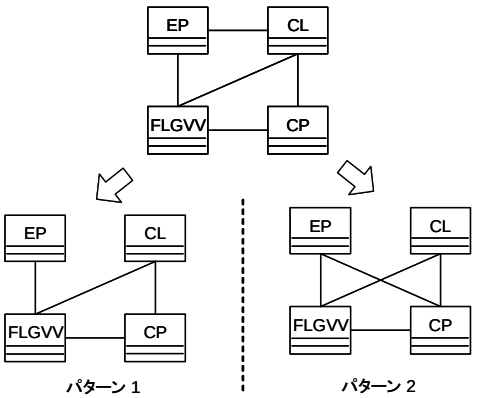

図 3．パターンごとのクラス間の結合変化

6. 議論

本ツールを使用することで，リファクタリングの中心的なクラス（今回の適用実験では，移動元・移動先クラス）以外に修正が必要なクラスを自動的に特定することができる．また，修正は加えないが CK メトリクスの値が変化するクラス（今回の適用実験では，EP クラス）を捉えることができ，影響を及ぼす箇所をすべて把握できるようになった．さらに，ソースコードの修正前にリファクタリング前後の CK メトリクスが得られることは，開発者が適切だと判断したリファクタリングがソフトウェアの複雑さの観点から有効であるかどうかを正確に捉えることができる．

また，今回の適用実験では，複数のリファクタリング候補の CK メトリクスを計測することで，それぞれ CBO に大きな違いが表れることが把握できた．開発者がソースコード上から，この違いが表れることを把握するのは極めて困難である．これらの違いをソースコード修正前に把握できると，開発者はリファクタリングパターンの選択において適切な判断が行えるようになる．

最後に，開発者からは「今回の適用実験のようにリファクタリングの候補がいくつか挙がった場合，ソースコードを修正せずにこれらの違いを知ることができるのはとても有益である」とコメントを頂いた．

次に，本ツールを使用し，2 通りのリファクタリングに関して適用前に CK メトリクスの計測を行ったが，実際にソースコードを修正したものと一致するかどうかを検証した．

パターン 1 を例に挙げ，ソースコードの修正の流れを具体的に説明すると，以下のようになった．

（1）CL クラスのフィールドおよびメソッド群を FLGVV クラスに移動する

（2）CLクラス内のsetStartClassメソッドの参照先を修正する（図 4）

（3）GraphViewPanel（以下GVP）クラス内のsetExtractedPanelメソッドの参照先を修正する（図 5）

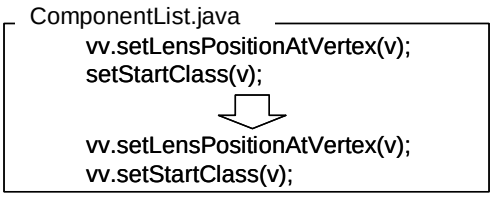

図 4．CL クラスの修正

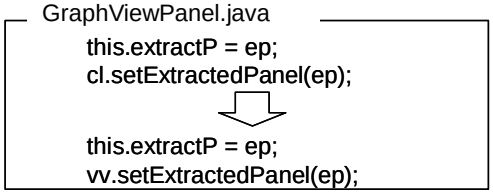

図 5．GVP クラスの修正

ソースコードを修正前と後で実際に CK メトリクスを計算した結果とツールを用いて計算した結果の比較を行ったところ，一致する結果が得られた．同様に，パターン 2 に関しても一致する結果となった．

しかし，実際は開発者ごとにコード規約が異なるため，ソースコードの修正は一意に決まらない場合が多く，機械的に変換を行うツールの結果と違いが表れる可能性がある．例えば，「クラス内部の変数へのアクセスは，getter/setter を介さずに直接参照する」といったコード規約があると仮定する．本ツールでは，今回の適用実験のようにクラス A のフィールドとその getter をクラス B へ移動するような場合，クラス B 内では getter を介してフィールドアクセスする変換が行われる．しかし，このコード規約に従いソースコードを修

正すると，getter を介さず直接アクセスに修正されてしまうため，結果に違いが表れる．

7. まとめ・今後の課題

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 メトリクスに基づき，リファクタリングが保守性に与える効果を計測する環境を実装した．また，研究室で開発中のツールを対象に適用実験を行い，各 CK メトリクスの変化に対する考察，本ツールの有用性の評価を行った．最後に，本ツールで計測した値と，実際にソースコードを修正した値に違いがないかを適用実験で使用した例を元に比較を行った．

また，今後の課題としては，以下のことがあげられる．

- リファクタリング前後で CK メトリクスの値が変わらない場合や値ごとに悪化・改善が見られ評価が難しい場合に対して，適切な評価軸を与える．
- 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．
- いまだ未実装なリファクタリングの実装を行い，それらの評価を行なう．

文　献

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

[2] S. Chidamber and C. Kemerer, A metric suite for objectoriented design, IEEE Transactions on Software Engineering, 25(5):pp.476–493, 1994.

[3] Y. Kataoka. T. Imai, H.Andou and T. Fukaya, A quantitative evaluation of maintainability enhancement by refactoring, in Proc. Int'l Conf. Software Maintenance., IEEE Computer Society, pp.576–585, 2002.

[4] B. Bois, S. Memeyer, and J. Verelst, Refactoring - improving coupling and cohesion of existing code, In Proc. the 11th IEEE Working Conference on Reverse Engineering, pp.144–151, 2004.

[5] 秦野，乃村，谷口，牛島．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，44(6):pp.1548− 1557, Jun 2003.

[6] T. Tourwe´ and T. Mens, Identifying Refactoring Opportunities Using Logic Meta Programming, Proc. European Conf. Software Maintenance and Reeng., pp. 91-100, 2003.

[7] Roberts, D. Brant, J. and Johnson, R., A Refactoring Tool for Smalltalk, Theory and Practice of Object Systems (TAPOS), Vol.3, No.4, pp.439-442, 1997.

[8] jFactory, http://www.instantiations.com/jfactor/

[9] JRefactory, http://jrefactory.sourceforge.net/

[10] Transmogrify, http://transmogrify.sourceforge.net/

[11] V. R. Basili, L. C. Briand, W. L. Mélo, A validation of object-oriented design metrics as quality indicators, IEEE Trans. on Software Eng., Vol. 20, No. 22, pp.751-761, 1996.

[12] E. J. Weyuker, Evaluating software complexity measures, IEEE Trans. on Software Eng., Vol.14, No.9, pp.1357-1365, 1988.

[13] 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，博士論文，大阪大学大学院基礎工学研究科 博士論文，2001.

[14] 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，電子情報通信学会技術研究報告，SS2006-67，Vol.106，No.427，pp.13-18，2006.

[15] Eclipse, http://www.eclipse.org/

[16] M. H. Halstead, Elements of Software Science, Elsevier North-Holland, 1977.

[17] McCabe, T.J., A complexity measure, IEEE Trans. Software Eng., vol. SE-2, no. 4, pp.308-320, 1976.

[18] Class Construction Kit, http://bcel.sourceforge.net/cck.html