不変情報を用いたリファクタリング支援

片岡欣夫[†,††] 楠本真二[†] 井上克郎[†]

リファクタリングはプログラムの可読性や保守性など，非機能品質の改善を目的としたプログラム変換技術である．しかし，リファクタリングすべき箇所の特定や適切なリファクタリングパターンの選択に多くの労力が必要となるため，実適用が非常に難しい状況になっている．本稿ではプログラム中から，特定のリファクタリングパターンが適用可能なリファクタリング箇所を自動的に検出するための手法を提案し，その有効性を示す．提案手法はプログラムの不変情報を利用しているところに特徴がある．具体的には，プログラム中のある地点で一定の不変情報のパターンが成立するとき，“Remove Parameter”，“Eliminate Return Value”，“Separate Query from Modifier”，“Encapsulate Downcast” のいずれかのリファクタリングパターンが適用可能であるという関係を利用する．さらに，これらのリファクタリングパターンに対応した不変情報パターンマッチングツールを開発した．最後に，提案手法を実際の Java プログラムに適用した結果，多くのリファクタリング候補とリファクタリングパターンが自動検出され，これらはプログラムの開発者から有効なリファクタリングであると評価された．

Supporting Refactoring Using Invariants

Yoshio Kataoka,[†,††] Shinji Kusumoto[†] and Katsuro Inoue[†]

Program refactoring — transforming a program to improve readability, structure, performance, abstraction, maintainability, or other features — is not applied in practice as much as might be desired. One deterrent is the cost of detecting candidates for refactoring and of choosing the appropriate refactoring transformation. This paper demonstrates the feasibility of automatically finding places in the program that are candidates for specific refactorings. The approach uses program invariants: when a particular pattern of invariant relationships appears at a program point, one of the following refactorings, “Remove Parameter,” “Eliminate Return Value,” “Separate Query from Modifier,” and “Encapsulate Downcast,” is applicable. Since most programs lack explicit invariants, an invariant detection tool called Daikon is used to infer the required invariants. We developed an invariant pattern matcher for several common refactorings and applied it to an existing Java code base. Numerous refactorings were detected, and one of the developers of the code base assessed their efficacy.

1. はじめに

リファクタリング[1] はソースコードの構造を改善し，以降の保守工程でのコストを削減することなどを目的として行われる．1990 年代の初めに提唱されてから，ソフトウェア開発の主流技術の 1 つとして認められるようになってきた．1 つの例としては，ソフトウェア開発中に定常的にリファクタリングを行うことが重要であるということが，eXtreme Programming (XP)[2] の主要な教義の 1 つとなっていることなどがあげられる．

学術的にはその重要性が認識されている[3),4)] にもかかわらず，企業の実際の開発現場でリファクタリングが適用されることは多くはない．リファクタリングを行う時間的余裕がないという管理的側面の問題や，リファクタリングを行うことによる不具合の危険性といった技術的側面の問題が，その理由としてあげられる．これまでにこれらの問題を解決するためのリファクタリング支援ツールがいくつか提案されてきている[5)〜9)] が，多くは開発者が適用すべきと判断したリファクタリングを自動的かつ安全に適用することを支援するものであり，リファクタリング候補の特定についてはほとんど支援されていない．品質改善に有効なリファクタリング候補を手作業で検出するためには，

† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
†† 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
System Engineering Laboratory, Corporate Research & Development Center, Toshiba Corporation

開発や保守を通じて現れる設計品質劣化を見定める必要がある．しかし，納期や仕様変更への対応に追われる現場のプログラマの立場からは，これらの設計上の問題点は見過ごされたり無視されたりする場合が多い．

そこで，本稿では，リファクタリング候補の自動検出手法を提案し，その支援ツールの開発を行う．リファクタリング候補の検出にあたっては，プログラムの不変情報☆を利用する．ある特定の入力に対してプログラムを実行し，その際プログラム上のある場所においてある一定の不変情報のパターンが発見されると，特定のリファクタリングパターンが適用できる可能性があることを示そうというアプローチである．具体的なリファクタリングパターンとしては，“Remove Parameter”，“Eliminate Return Value”，“Separate Query from Modifier”，“Encapsulate Downcast” の 4 種類を対象としている．次に，提案手法に基づいたリファクタリング候補自動検出ツールを開発した．開発したツールでは，不変情報の検出に動的不変情報検出ツールである Daikon[10] を用いた．さらに，実際の Java プログラムに対するケーススタディを行った．結果として，適用した Java プログラムの開発者から提案手法によって特定されたリファクタリング候補とリファクタリングパターンの妥当性が確認された．

以降，3 章ではリファクタリングや不変情報の検出に関するこれまでの研究について述べる．次に，4 章で提案するリファクタリング候補検出方法について説明し，5 章で提案手法の有効性を評価するために行ったケーススタディについて述べる．最後に，6 章でまとめと今後の課題について述べる．

2. リファクタリングに対する開発現場の反応

以下では筆者のこれまでの知見などをもとに，リファクタリングに対する開発現場の反応を紹介する．

前述のように，企業の開発現場ではリファクタリングが積極的に，あるいは組織的に行われているとはいい難い状況にある．なぜリファクタリングという技術が適用されないかについては，以下のような理由があげられる[11]．

- 開発納期の問題
  開発現場では要求された機能をいかに実現するかということが最優先の課題となっている．リファクタリングを行うためには，ソースコード改変に付随するすべての開発作業（ソースコードの登録，リビルド，各種テストなど）を一通りやり直す必要がある．大部分のプログラマやプロジェクト管理者にとってリファクタリングは機能開発後に発生する付加的な作業であるという認識が根強く，「本質的でない作業のために工数を割くわけにはいかない」といった理由でリファクタリングの適用が見送られるケースが多い．
- バグ埋め込みの危険性
  前項の理由と関連するが，すでに動作をして試験をパスしているプログラムに対して改変を加えるという行為に強く抵抗を覚えるプログラマや管理者が大多数を占める．テストファーストプログラミング[2] などのプログラミングパラダイムは開発現場にはあまり浸透はしていないので，ソースコードを改変したときに品質保証をする手段として，テスタに作業を委ねることが唯一安全な方法であると認識しているプログラマや管理者がほとんどである．
- リソース不足の不安の低減
  たとえば一昔前の組み込みソフトウェア開発では，限られた実装メモリ制約のなかでいかに効率的に実装を行うかということが非常に重要視されており，実装が進む段階でも開発と並行したリファクタリングの実施はある意味当然のように行われていた．しかしながら昨今携帯電話のような非常に小さなデバイスでさえ潤沢なハードウェアリソースが利用可能になっており，実装に対する制約条件が非常に緩くなってきている．結果として保守性や拡張性といった目に見えない形での要請の重要性は，開発者やプロジェクト管理者には理解され難くなってきている．
- 開発規模の増大
  ソフトウェア開発は大規模化と多機能化，開発サイクルの短縮といった流れにともない，外注，孫外注といった二重三重の開発委託体制が当たり前のようになってきている．複数の開発チームに別れた 100 人規模の開発者を統一的に管理することは非常に困難であり，個々のチームの開発の仕方に細かな指示を与えることは難しく，出口管理が唯一の品質保証の手段であるといった状況に陥りがちである．このことはさらに，システム全体のアーキテクチャの統一を図ることが非常に困難であるという問題を生み，皮肉なことにリファクタリングの必要性をさらに高める結果となっている．

もちろん以上あげた以外の様々な要因が存在しうるが，いずれのケースにも共通しているのは以下の 2 段

☆ “invariant.” 不変表明，不変条件とも訳されるが，本稿では不変情報という訳語をあてる．

構造である．

(1) リファクタリングの有効性は認識されている．リファクタリングを行うことで，たとえばシリーズ製品の開発の際に重要な再利用性が確保できたり，アーキテクチャの維持が容易になったりすることは理解されている．特に昨今ではソフトウェアアセットの再利用は当たり前の要請として存在するので，高品質で再利用性の高いソフトウェアを生産するための技術としてリファクタリングが有効であるということに異論を唱える現場の担当者はいない．

(2) 一方保守性や拡張性といった非機能品質の確保が，機能品質の確保に比べてきわめて優先度が低い．別のいい方をすれば，リファクタリングを行うためのコストが得られる効果に比して高すぎると認識されている．

この二律排反を解消するために，リファクタリングにかかるコストを削減するというアプローチが 1 つの解として考えられる．次章以降ではこれまでの研究成果と我々のとったアプローチについて順に議論を進めていく．

3. 関 連 研 究

3.1 リファクタリングツール

理想的にはリファクタリングは自動的に適用されるのが望ましい．しかし，一般的なソフトウェアに対する変更と同様に，開発者が人手でリファクタリングを適用する場合には誤りが入り込む可能性がある．本稿では特にリファクタリング候補の検出，さらには人手で適用されたリファクタリングが正しく意味を保存しているかどうかを確認するための技術に焦点を当てる．一方，これまで報告されているほとんどのリファクタリング関連の研究では，リファクタリング箇所に対するリファクタリングパターンの自動適用技術に焦点を当てている．これらの 2 つの技術は，お互いを補完しあうものである．

Opdyke と Griswold はリファクタリングを自動的に適用するためのツールを紹介している[5),6)]．

また，リファクタリングを支援する機能を持ったいくつかのツールが提案されている．Smalltalk Refactoring Browser[7)] は，Opdyke の初期の研究で取り上げられているリファクタリング[12)] を自動的に適用するためのツールである．IntelliJ IDEA（www.intellij.com）は，Java のパッケージや変数の名前の付け替えや移動を支援するためのツールである．また XrefSpeller（www.xref-tech.com/speller/）は，Emacs の拡張パッケージであり，C や Java に対するいくつかのリファクタリングを支援する．

Roberts[6)] は，Opdyke の定義したリファクタリングはそのままで適用可能なものはほとんどないことを述べ，リファクタリングが適用された後に成立すべき後条件を定義した．この定義により，リファクタリングに関する前条件と後条件の依存関係の詳細な推論が可能になり，複合的なリファクタリングの定義が可能になった．さらに，Roberts は動的なリファクタリングに関しても議論している．動的なリファクタリングとは，プログラムの動作中にある特性を検査したうえで適当なリファクタリングを適用するものであるが，守られているべき条件が破られたことが確認された時点で適用したリファクタリングを元に戻すことができるようになっている．動的解析によって得られる述語論理を調べることによってリファクタリングを支援しようとする点において，Roberts の研究は我々の成果と類似しているといえる．しかし，彼の研究は上述のツールと同様にリファクタリングの適用に焦点を当てており，リファクタリングを適用すべき場所を特定する我々の研究とはその点で異なっている．

Moore の Guru ツール[8),9)] はやや大域的な特定の 2 つのリファクタリングの自動化を行う．Guru はグラフ理論に基づいた継承階層の推論アルゴリズムを採用しており，Self[13)] で書かれたプログラムのオブジェクト階層を自動的に再構成することができる．またメソッド間の共通式の括り出しも，同様の手法を用いて自動化されている．

Bowdidge らの Star Diagram[14),15)] は選択された変数やデータ構造に対する参照を階層的に分類し，木構造としてグラフィカルに可視化することで冗長な構造を見出すことで適切なオブジェクト指向的再設計を促進する．この可視化手法は自動リファクタリングツールおよび refactoring planner[16)] のフロントエンドとして採用されている．ただし，ユーザが自ら適用候補となる変数やデータ構造を指定する必要がある．

3.2 コード中の冗長部分の発見

ソフトウェアシステムに潜在するコードの複製を発見する技術も，関連研究としてあげられる．複製されたコード片は，それらをまとめて 1 つのモジュールに集約する（“Extract Method” や “Pull Up Method” など）といったリファクタリングの候補となりうる．

Baker[17)] は，コード部分を文字列比較することでソフトウェアシステム内の重複コードを検出する技術について論じている．また，Kontogiannis ら[18)] はソースコード中のパターンを見つけるために，ソースコー

ドメトリクス，動的プログラミング，および静的マッチングといった 3 つの手法について述べている．いくつかの Unix シェルなど中程度の規模のシステムに対しては，実験の結果有効な手法であるということが確認されている．

一方，類似コード片に適用可能なリファクタリングパターンは，不変情報により検出されるリファクタリング候補には適用できない．その意味で，我々が提案する手法とは補完関係にある．

4. 提案手法

本研究の最終目的は，開発者がリファクタリングを動機付けるような 20 種余りの “bad–smell”[1] を発見することを支援することである．発見されたリファクタリング候補を実際に適用すべきかどうかに関しては，依然人手による判断が必要である．当該リファクタリングを適用するだけの価値があると判断されれば，人手によるなり 3 章で紹介されたようなツールを使うなりして，実際にリファクタリングを行うことになる．

リファクタリング候補を発見するためには，何らかの形でプログラムの特徴点を抽出する必要がある．“Large Class”[1] のようないくつかの場合には，特徴点抽出手法として各種メトリクスを利用するなどで事足りるが，一般的にはプログラムの意味的解析が必要となる．我々はプログラムの意味的側面を反映する不変情報を利用することにより，その特定のパターンを特徴点としてリファクタリングと対応づけることにより，適用可能なリファクタリング候補を検出する手法を提案する．

我々の手法が対象とするのは，機能実装がほぼ収束した開発終盤におけるリファクタリングである．その理由の 1 つとして開発中盤までは有効な不変情報が得られる保証が少ないという制約があげられる．別の理由としては，前述したようなアーキテクチャの維持などを目的としたリファクタリングを実施する場合には，なるべく広い範囲を対象としたリファクタリングを優先的に実施することが有効であるというものがあげられる．ただしこのことは，我々の手法が開発の中盤以前に行われる可能性のあるリファクタリングと相容れないということではなく，開発フェイズにあったリファクタリングを組み合わせることでより効果的に働くと考えている．

4.1 概　　要

我々の手法の概要を**図 1** に示す．

提案手法は，不変情報の検出と不変情報に対するパターンマッチングから構成される．検出した不変情報に対して，いくつかのリファクタリングパターンに適用可能になる不変情報のパターンとのマッチングをとり，リファクタリング候補と適用可能なリファクタリングパターンを出力する．

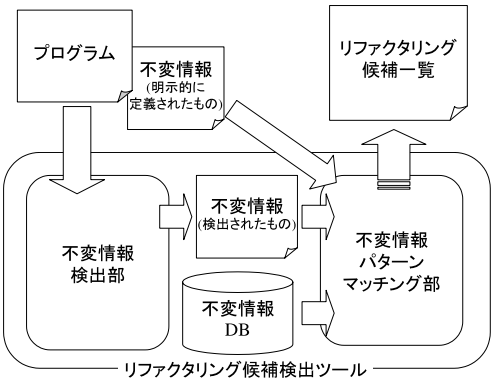

図 1 リファクタリング候補検出手法の概要
Fig. 1 Overview of refactoring candidate detection.

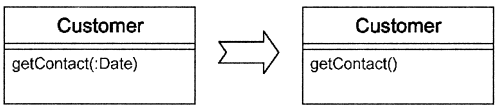

図 2 Remove parameter
Fig. 2 Remove parameter

なお，不変情報は，プログラム中で明示的に与えられる場合と与えられない場合がある．不変情報が明示的にプログラム中に与えられているような場合は，それらをリファクタリング候補を検出するための情報として利用する．明示的な不変情報がほとんど，あるいはまったく利用できないような一般的な場合には，既存の不変情報検出手段を利用する．4 章で述べるケーススタディでは，不変情報検出に動的不変情報検出ツール Daikon[10] を用いている．

一般的には対象とするリファクタリングごとに必要な不変情報の粒度は異なるため，どのような粒度の不変情報が利用可能か，あるいは検出可能かということにより，本手法で対応可能なリファクタリングの種類は限られる点には注意が必要である．たとえば Daikon ではメソッド単位，クラス/インスタンス単位での各種不変情報が検出可能であり，以降紹介する事例では各々特定の粒度の不変情報を用いてリファクタリング候補の検出が行われている．

以下では我々のツールで検出可能なリファクタリング（多くは Fowler ら[1] からの抜粋）を紹介し，それらの適用可能性を示唆する不変情報について述べる．

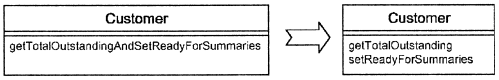

図 3 Separate query from modifier
Fig. 3 Separate query from modifier.

4.2 不変情報から検出可能なリファクタリング候補

4.2.1 Remove Parameter

'Remove Parameter' は,「引数がメソッド本体で利用されていない場合」に適用することを目的としたリファクタリングである[1)]．引数の値がつねに一定の値しかとらない場合や他の情報から計算可能な場合でも，その引数は削除することができる．このリファクタリングは，メソッドの定義文とすべてのメソッド呼び出しから，当該引数を削除することになる．このリファクタリングを施すメリットとしては，余計な引数が存在することでコードの利用者に与える混乱や負担を軽減することができるということがあげられる．

あるメソッド M について 'Remove Parameter' が適用可能である，すなわち特定の引数 p が削除できるのは，不変情報の観点で以下の前条件が成立している場合である．

- p が定数である．
- $p = f(a, b, \cdots)$ である．

ただし $f()$ を任意の関数，$a, b, \cdots$ を M の引数，あるいは p と同一スコープ内の任意の変数とする．

4.2.2 Eliminate Return Value

このリファクタリングは Fowler ら[1)] の中では述べられていないが，そのメリットや仕組み自体は 'Remove Parameter' と同様である．'Eliminate Return Value' は，単純な値を返しているようなメソッドや，呼び出し側でつねに無視されるような値を返しているだけのメソッドを対象としている．単純な値を返している場合というのは，その返り値がつねに一定の値であったり他のスコープ内の情報から一定の式で計算可能であったりする場合である．

あるメソッド M について 'Eliminate Return Value' が適用可能である，すなわちその返り値 r が削除できるのは，不変情報の観点で以下の後条件が成立している場合である．

- r が定数である．
- $r = f(a, b, \cdots)$ である．

ただし $f()$ を任意の関数，$a, b, \cdots$ を呼び出し側のスコープ内の任意の変数/定数とする．

4.2.3 Separate Query from Modifier

'Separate Query from Modifier' は,「返り値を返しかつオブジェクトの状態を変化させるようなメソッド」に対して適用するべきリファクタリングである[1)]．本リファクタリングは単一の処理を，問合せ（query）に対して値を返すだけの部分と，オブジェクトの内部状態の変更を行う部分（modifier）の 2 つの処理に分けることになる．このリファクタリングを行うことにより，単一のメソッドに対して単一の目的を与えることができ，メソッドを利用する側からすれば問合せだけ，あるいは内部状態の変更だけという利用が行えるようになるというメリットがある．これらのメソッドは単一の目的を果たすため，他の処理に組み込む場合にも副作用の心配をする必要がなくなる．こういった副作用については，メソッドの名前やコメントを見ても明示的に記されていないケースも多く，このことからこの手の問題をあらかじめ検出するのは重要であるといえる．

```
Object lastReading() {
  return readings.lastElement();
}
```

```
Reading lastReading() {
  return (Reading)readings.lastElement();
}
```

図 4 Encapsulate downcast
Fig. 4 Encapsulate downcast.

あるメソッド M について 'Separate Query from Modifier' が適用可能である，すなわち M を問合せ部 M_q と内部状態変更部 M_m という 2 つのメソッドに分解できるのは，M の出口において不変情報の観点で以下の 2 つの条件が成立している場合である．

- 返り値があり，かつ後条件に「返り値が定数である」という不変情報が含まれていないこと．
- メソッド入り口でスコープに入っているある変数 a（たとえば引数やインスタンス変数など）に関して，後条件に $a \neq orig(a)$，あるいはそれを含意する不変情報が含まれていないこと．

仮に「返り値が定数である」という条件が含まれているならば，先に述べた 'Eliminate Return Value' が適用できる．$orig(a)$ は変数 a のスコープ入口での値を意味する．すなわち $a \neq orig(a)$ は,「変数 a が元の値から変化している」ということを示している．つまり $a \neq orig(a)$ を含意する不変情報とは，たとえば $a = func(a)$（$func()$ は任意の関数）などがあげられる．

4.2.4 Encapsulate Downcast

'Encapsulate Downcast' は,「呼び出し側でのダウ

ンキャストが必要な返り値を返しているメソッド」に対して適用すべきリファクタリングである[1)]．本リファクタリングは返り値の型を変更して，ダウンキャストをメソッドの本体側に隠蔽する．これによって静的なダウンキャストの個数を減らすことができ，メソッドを利用する側の実装を簡便にし，理解性を高めることができる．また動的な型チェックに頼ることなく，静的に，すなわちコンパイル時に型チェックが行えるようになるため，不具合の早期発見と実行時の効率向上という二重の利点が得られる．

あるメソッド M について 'Encapsulate Downcast' が適用可能である，すなわち呼び出し側で必要となっているダウンキャストを本体側に隠蔽することができるのは，M の出口において不変情報の観点で以下の条件が成立している場合である．

- $LUB(return.class) \neq declaredtype(return)$

ここで $LUB()$ は最大下界を与える関数で，$LUB(return.class)$ は，返り値のクラス集合の最大下界を表す．また $declaredtype(return)$ は，返り値の宣言クラスである．返り値のクラス集合の最大下界が宣言クラスに等しくないということは，すなわちすべてのケースでダウンキャストが行われているということである．

5. ケーススタディ

提案する不変情報のマッチングによるリファクタリング候補検出手法の有効性を評価するためのケーススタディを実施した．対象プログラムは，Aspect Browser[19)] のコンポーネントである Nebulous を対象とした．

その理由としては，(1) 我々と開発者との間に交流があり，抽出したリファクタリング候補の妥当性を実際に評価してもらうことが可能であること，(2) Nebulou の開発は現在も継続して行われており，リファクタリングを行うべき箇所の存在が予想できること，(3) ある程度大規模なプログラムであること（約 7,000 行），があげられる．

また，Nebulous から不変情報を検出するために，不変情報検出ツールとしてよく知られている動的不変情報検出システム Daikon[10)☆] を利用した．

以下では Daikon の紹介と，Nebulous を対象として行ったケーススタディの概要と結果について述べる．

☆ Daikon は，`http://pag.csail.mit.edu/daikon/` からダウンロード可能である．

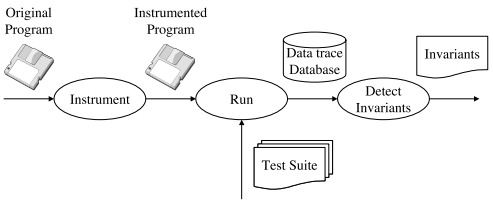

図 5 Daikon の処理の流れ
Fig. 5 Process flow of Daikon.

5.1 Daikon：不変情報検出ツール

Daikon は，プログラムを実行することで動的に不変情報の候補を検出する．処理の流れを**図 5** に示す．

まず入力プログラムに計測対象とする変数の値の変化を追跡するためのプローブを挿入する（Instrument）．次にそのプログラムを実行し（Run），得られたログを解析して不変情報を推論する（Detect Invariants）．不変情報の推論では，存在する可能性のある不変情報の候補をログに残されたデータを用いて検証していく．ある一定の基準を満たして，存在の可能性が否定されなかった不変情報の候補が，ユーザ側に当該プログラムの不変情報として提示される．テストやプロファイリングといった他の動的手法と同様に，推論された不変情報の正当性は，入力として利用されるテストケースの品質や完全性に左右される．不変情報検出システム Daikon は言語非依存であり，現在のところ C/C++，Java，および Lisp をサポートしている．

不変情報推論部について，詳細に説明する．Daikon はプログラム上のある地点（たとえば，ループ構造の先頭や手続きの出入り口）に対応する不変情報を，それぞれ独立に検出することができる．不変情報推論部には，プログラム上のある地点においてスコープ内にあるすべての変数の値が記録されたトレースログが与えられる．不変情報の候補は，ログに記録された単一の変数の値，および 2 つまたは 3 つの変数の組合せに対して検査される．

たとえば，変数 x，y，z，および定数 a，b，c が存在する場合，以下の (1)〜(9) に示されるような不変情報候補が検査される．

(1) 定数との等価関係（$x = a$）
(2) 定数の集合（$x \in a, b, c$）
(3) 範囲（$a \leq x \leq b$）
(4) 非零
(5) 剰余（$x \equiv a(mod b)$）
(6) 線形関係（$z = ax + by + c$）
(7) 順序関係（$x \leq y$）
(8) 特定の関数（$x = (y)$）

(9) 上記の複合形（$x + y \equiv a(mod\ b)$）

また配列などに関しての不変情報も検査される．たとえば配列中の最大値，最小値，辞書的順序関係，数値的順序関係など，あるいは全要素で成立している不変情報，あるいは集合の包含関係などが対象となる．例として 2 つの配列が与えられた場合，要素ごとの線形関係，辞書的順序による比較，さらには一方が他方を含む部分列の関係が検査される．

なお，特定の変数について十分なサンプルが得られなかった場合は，そこで得られたパターンは単なる偶然の結果にすぎないことも考えられる．そこで，Daikon では不変情報は適当なレベルで妥当性が確認されて初めて出力される．具体的には，検出された各不変情報について，任意の入力に対して検出されたような特性が偶然発現する確率を計算し，その値がユーザがあらかじめ定義した確信度のパラメータを下回ったものだけを出力するようにしている[20]．

以上紹介した Daikon の機能を利用することで，4.2 節であげた 4 つのリファクタリングパターンに対応する不変情報を検出することが可能になる．“Remove Parameter” のケースを例にとり，対応する不変情報がどのように検出されるかを見てみる．前述のとおり，あるメソッド m における特定の引数 p に対して，“Remove Parameter” が適用できるための不変情報は，以下のいずれかであった．

- p が定数である．
- $p = f(a, b, \cdots)$ である．

前者に関しては，たとえば $p = 5$ といった不変情報が検出されていれば，p はつねに一定の値を渡すためだけに使用されているということが分かる．また Daikon が実行ログからデータマイニングによって検出する上記 (1)〜(9) の各種変数間の関係式の中に p に関連する不変情報が含まれていれば，後者の条件が満たされていることになる．

5.2 対象プログラム：Nebulous

Nebulous は簡単なパターンマッチング機構と地図のメタファを用いて，プログラムの特徴や特性がファイル階層中をどのように横断しているのかを可視化し，それら特徴や特性に対する変更を管理するためのツールである．Nebulous は Java 言語で記述され，78 個のクラスで構成され，サイズは約 7,000 行である．

5.3 実験手順

(1) 特定のリファクタリングパターンが適用可能であることを示す不変情報のパターンを検出するための Perl スクリプトを準備する．
(2) 典型的な Nebulous の実行パターンを通して，Daikon を用いた不変情報の検出を行う．具体的には，Nebulous の様々な機能を対象にテスト実行を行う．
(3) 検出された不変情報に対して (1) の Perl スクリプトを適用して，リファクタリング候補を検出する．
(4) Nebulous の開発者に検出されたリファクタリング候補とリファクタリングパターンを提示し，その有用性を評価してもらう．

表 1 実験結果
Table 1 Result.

リファクタリング	yes	maybe	no	計
Remove Parameter	6	4	5	15
Eliminate Return Value	1	2	4	7
Separate Query from Modifier	0	2	0	2
Encapsulate Downcast	1	1	0	2
合計	8	9	9	26

Nebulous の開発者は，検出されたリファクタリング候補を以下の 3 種類に分類した．

yes 候補は有用なリファクタリングである．
no 候補は有用なリファクタリングではない．
maybe 候補は条件次第では有用なリファクタリングであるか，あるいは別のリファクタリングの方が有用である．

3 つ目の ‘maybe’ の意味は，検出された候補は確かに保守性に影響を与えうる場所を特定しているが，緊急には保守を要さないためにリファクタリング適用コストを考慮すると放置していても問題がないと判定されたケース，あるいは，保守性の問題は確かに存在するが，今回対象としたリファクタリング以外の手段で対策を講じた方が効果的であると判定されたケースがこれにあたる．

5.4 実験結果

実験結果を**表 1** に示す．

5.4.1 Remove Parameter

6 個の ‘yes’ のうち 5 個が，メソッドの呼び出し側から同じ値やオブジェクトが渡されているというものであった．オブジェクトが引数として渡されているケースでは，オブジェクトに含まれるすべてのデータが渡らなければならないため，単純な変更ではすまない．オブジェクトはシングルトンであるため，これを静的なクラスにするようなリファクタリングが適当である．結果として，目的のデータに対する静的なアクセスメ

ソッドを定義でき，引数の除去が可能になる．

‘no’ と ‘maybe’ のほとんどのケースは，フラグを引数として渡しているときにすべての呼び出しで同一の値が渡されており，かつ，このフラグがメソッド内で手続きの分岐に用いられている場合であった．この結果を受けた開発者は “Remove Parameter” を実行することは適当ではないと判断したが，“Explicit Methods” を施しつつ，“Replace Parameter” を実施する[1] ことが適当であろうと結論づけた．これによって分岐のロジックをメソッド外に除去することが可能になり，メソッドの保守性が向上すると期待される．このようなケースに対応するように，たとえば，引数として渡される値が一定の範囲にあることを発見する手段などと組み合わせて，フラグとして扱われる引数を判別するようにツールを拡張すれば，より適切なリファクタリング候補の検出ができるようになると考えられる．

5.4.2 Eliminate Return Value

ここでの ‘yes’ はつねに真を返すメソッドを正確に特定したものであった．

一方，4 つある ‘no’ は，入力に用いたテストデータが Nebulous の持つ，より深い機能を実行するに不十分であったために発生したものであった．この原因は，今回不変情報検出手法として動的手法を用いたことが考えられる．すなわち不変情報として検出されたものは，実際はより多くのテストデータを検査することで排除されるべきものであった．2 つの ‘maybe’ は単純に機能的な観点からは正しいリファクタリング候補であるが，実際の価値としては疑問が残るというケースであった．つまり，指摘されたメソッドは現時点では，返り値を返す必要がないが，将来的に何らかの追加処理を行う可能性があるため，返り値を返す仕様を残しておく方が良いと判断した．

5.4.3 Separate Query from Modification

2 つの ‘maybe’ はプログラミングスタイルに関するものであった．たとえば，一般に，変更して値を返すイテレータパターンを好むプログラマならば，必然的に問合せと変更を組み合わせたようなメソッドを実装する．素直にこのような場合を取り扱おうとすれば，単純にプログラマの好みによって不変情報パターンの検出ルーチンをカスタマイズできるようにしておけばよく，プログラミングスタイルと矛盾をきたすようなリファクタリングを推奨しないようにすることができる．

5.4.4 Encapslate Downcast

ツールの検出した 2 つの候補はいずれも正しいリファクタリングであった．そのうち一方はあるベクトルに対する合計 10 個のキャストが関与していたが，他方に関しては合計 2 個のキャストが存在しただけであった．開発者は後者のリファクタリングについては ‘maybe’ の評価を与えた．これは新たなクラスを導入してダウンキャストをカプセル化するコストの方が，2 つのキャストを解消する利点を上回ってしまうとの判断による．

5.5 分析と考察

全体として，今回評価を行った開発者は提案手法によるリファクタリング候補の抽出が有用であるという評価をした．ツールが検出したリファクタリング候補は，数的にはそれほど多くはなかったものの，Nebulous の基本的なアーキテクチャ上の問題点を指摘したものであった．たとえば Nebulous におけるいくつかのフラグを用いたメソッド分岐の例が，リファクタリング候補として検出されていた．これは数多くある同様のパターンのうち，使用したテストデータに関連した一部を検出しただけであるが，開発者は自ら持っている Nebulous の背景知識をもとにこれをアーキテクチャレベルの問題に一般化してとらえることができた．またいくつかのリファクタリング候補は開発者には興味のないものであったが，示された候補が重要か重要でないかの判断は非常に短時間に行うことができ，無駄な提案に余計な時間をとられることもなかった．さらには ‘no’ と判断された候補についても，Nebulous のアーキテクチャに対する新たな洞察を与える場合があった．

検出されたリファクタリング候補のいくつかは，正しいが適用するには問題の残る微妙なものも存在した．たとえば “Remove Parameter” では，シングルトンを静的なクラスに変更をしなければならない場合が起こりうる．別の例として，検出されたリファクタリングを精査することにより，例外処理などをなくしてプログラムを簡略化できることを確認できた場合がある．この場合リファクタリングの定義でいうところの「機能を変更しない」という要件に抵触するため，単純なリファクタリングではなく機能の再設計をともなう変更が必要になる．

これらの結果を受けて，開発者はアーキテクチャレベルの問題点を解決することを計画し，特にフラグの過度の利用を避けるようなリファクタリングを行って，いくつかのシングルトンオブジェクトを静的クラスに変換するようにした．

開発者は偶然にも少し前にテキストのパターンマッチを利用したコードクローン検出ツール[21] を使って，コ

ピーペーストされたコード片を洗い出そうとしていた．その結果，“Extract Method” や “Pull up Method” といった，共通要素を括り出すタイプのリファクタリング候補を中心にいくかの候補を検出することができた．しかし一方で，我々の手法で検出しえたリファクタリング候補については，コードクローン検出法では発見できず，両手法で検出された候補は大部分が直交するものであることが確認された．これは我々の手法がテキストベースの静的解析では利用できない変数の値などを参照して分析していることから期待された結果ではあるが，いずれにしても本手法が価値のあるリファクタリング候補を検出できているという 1 つの傍証になっている

また，いくつかの候補が ‘maybe’ と判断されたのはその内容の正確性で劣るからではなく，設計改善の観点からのリファクタリングの有用性の度合いに関係するということも確認できた．このような場合についてはユーザに無用な判断をさせないように，あらかじめ検出するリファクタリング候補を限定させるような洗練されたインタフェースを準備することでツールの有用性を高めることができる．今回扱ったそれ以外のリファクタリングに関しては，‘maybe’ と判断されたものは有用さの度合いに関連する．

たとえば “Encapsulate Downcast” では，ユーザは問題となっているキャストがいくつあるか，またどの程度広く分布しているかということを考慮して，提案されたリファクタリングの有用性を判断していた．キャストの数や分布は，式がどのように表現されているか，あるいはクラスやメソッド中にどのようにモジュール化されているかということに関連することから，プログラム設計の静的な属性としてとらえられる．

さらに “no” と判断されたもののうち，“Eliminate Return Value” と “Remove Parameter” に関しては，テストケースの偏りが原因でそれぞれ返り値や引数が定数であると判断された結果検出されたものが含まれていた．これは Daikon が動的解析をベースに不変情報を検出するがゆえの潜在的な不完全性であり，投入するテストケースを増強したり最適化したりすることである程度は解消可能であると考えているが，Daikon 単独では本質的な解決は困難である．

これらの問題への対処法としては，今回用いた動的解析技術を補完するために，何らかの静的解析技術を組み合わせることによって，これらのリファクタリング候補の検出精度をより高めることが可能であると考えられる．

6. おわりに

本研究では不変情報を用いて効率良くリファクタリング候補を検出する手法を提案し，その有効性を確認した．評価にあたっては，比較的大規模な実プロジェクトのソースコードに対して不変情報の検出を行い，それらを用いてプロジェクトの開発担当者がそれまで見つけることのできなかったリファクタリング候補を検出することができた．今回の結果から，以下の 2 点を確認できたと考えている．(1) 不変情報の検出と不変情報に基づくリファクタリング候補の検出もある程度自動化が可能である，(2) 提案手法が不変情報という動作に関する特徴によって裏付けられたリファクタリング候補を検出していることから，プログラムの作用を変更せずに構造だけを変更するという観点で，正しいリファクタリング候補を検出していることが保証されている．もちろん，今後も多くの適用事例を通して，これらの点を検証していく必要がある．一般的なソフトウェアプロジェクトを考えたときに，今回の適用例のみで十分であるあるとはいえない．提案手法がどのような条件のときに，有効に働くか，あるいは，働かないかということを確認することが重要である．

また，不要なリファクタリング候補の削除も必要である．あまり重要でない，あるいは意味のないリファクタリング候補を検出しないようにして，ユーザの負担を下げる必要がある．このためには，より効果的なテストケースを選んで不変情報の検出を行う，あるいは，静的解析技術を適切に組み合わせていくことが有用であると考えている．

さらに，提案手法が他のリファクタリング候補検出手法とどの程度うまく適合するのかということを確認する必要もある．原理的に我々の手法は，コードクローンベースの他の一般的な手法とは非常に異なった観点からのリファクタリング候補を検出することは明らかである．また実際に Nebulous の事例でもこのことは確認されており，一般に他の手法との相性は良いと考えている．また今回は 4 つのリファクタリングパターンを対象に検討を行ったが，他にも本アプローチが有効に働くであろうパターンが存在する．簡単な例を示すと “Inline Temp” というパターンは，一度しか参照されていないような一時変数をインライン展開してしまうというパターンであるが，これは静的な解析だけでは実際に一度しか参照されていないか，複数回参照される可能性があるのか，あるいはその場合どれくらい参照されるのかは不明である．このような場合不変情報による検出が有効に働くと考えられる．今

後さらに多くのパターンを精査して，どのパターンで syntactic な解析と semantic な解析をどの割合で組み合わせれば効率的な候補の検出が可能かということを調べていく必要がある．

リファクタリング候補の検出にプログラムの不変情報を利用するというアプローチは，相補的な関係にあるリファクタリング自動化手法やツールとあわせて，リファクタリング技術応用に大きなめどをつけたと考えている．さらには開発者が検出されたリファクタリング候補を通して，より大規模なアーキテクチャ的な改善に目をつけることができたという事実も重要である．2 章に述べたように，開発規模の増大にともないアーキテクチャの統一や維持が非常に困難になりつつある．低コストでアーキテクチャ上の問題点を指摘できるようになれば，頻繁なアーキテクチャのモニタリングや改善が可能になることが期待される．これはすなわち我々の提案したような手法なりツールなりが，ソフトウェアの改善や保守を通して起こりうる様々な問題に立ち向かうために非常に有効に働きうるという可能性を示したといえよう．

これまで企業の開発現場では敬遠されがちであったり，場合によってはタブー視されることも多かったリファクタリングという技術が，本研究の成果によって活用の機会が広がることが期待される．今後は上述したような拡張もあわせて検討し，企業におけるソフトウェア品質向上の一助となることを目指してゆきたい．

参 考 文 献

1) Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).
2) Beck, K.: *eXtreme Programming eXplained: Embrace Change*, Addison-Wesley (2000).
3) Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K. and Katoaka, Y.: On Refactoring for Open Source Java Program, *Proc. METRICS2003*, Sydney, Australia, pp.246–256 (2003).
4) 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，Vol.44, No.6, pp.1548–1557 (2003).
5) Griswold, W.: Program Restructuring as an Aid to Software Maintenance, Ph.D. Thesis, University of Washington, Dept. of Computer Science & Engineering, Seattle, WA (1991).
6) Roberts, D.: Practical Analysis for Refactoring, Ph.D. Thesis, University of Illinois at Urbana-Champaign (1999).
7) Roberts, D., Brant, J. and Johnson, R.: A refactoring tool for Smalltalk, *Theory and Practice of Object Systems*, Vol.3, No.4, pp.253–63 (1997).
8) Moore, I.: Automatic Restructureing of Object Oriented Programs, Ph.D. Thesis, University of Manchester (1996).
9) Moore, I.: Automatic inheritance hierarchy restructuring and method refactoring, *Proc. 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA*), pp.235–250 (1996).
10) Ernst, M., Cockrell, J., Griswold, W. and Notkin, D.: Dynamically discovering likely program invariants to support program evolution, *IEEE Trans. Softw. Eng.*, Vol.27, No.2, pp.1–25 (2001).
11) 片岡欣夫，本間昭次，深谷哲司：リファクタリングによる品質改善と品質劣化の予防 (1) リファクタリングプロセスの定義，情報処理学会第 63 回全国大会 (2001).
12) Opdyke, W.: Refactoring Object-Oriented Frameworks, Ph.D. Thesis, University of Illinois at Urbana-Champaign (1992).
13) Ungar, D. and Smith, R.: Self: The power of simplicity, *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.227–241 (1987).
14) Bowdidge, R.: Supporting the Restructuring of Data Abstractions throughgh Manipulation of a Program Visualization, Ph.D. Thesis, University of California, San Diego, Department of Computer Science & Engineering (1995).
15) Bowdidge, R. and Griswold, W.: Supporting the restructuring of data abstractions through manipulation of a program visualization, *ACM Trans. Software Engineering and Methodology*, Vol.7, No.2, pp.109–157 (1998).
16) Griswold, W., Chen, M., Bowdidge, R., Cabaniss, J., Nguyen, V. and Morgenthaler, J.: Tool support for planning the restructureing of data abstractions in large systems, *IEEE Trans. Softw. Eng.*, Vol.24, No.7, pp.534–558 (1998).
17) Baker, H.: 'Use-once' variables and linear objects – storage management, reflection and multi-threading, *SIGPLAN notices*, Vol.30, No.1, pp.45–52 (1995).
18) Kontogiannis, K., DeMori, R., Merlo, E. and Galler, M., et al.: Pattern matching for clone and concept detection, *Automated Software Engineering*, Vol.3, No.1–2, pp.77–108 (1996).
19) Griswold, W., Yuan, J. and Kato, Y.: Exploiting the map metaphor in a tool for software evolution, *Proc. 2001 International Conference on Software Engineering* (2001).

20) Ernst, M., Czeisler, A., Griswold, W. and Notkin, D.: Quickly detecting relevant program invariants, *Proc. 22nd International Conference on Software Engineering*, Limerick, Ireland, pp.449–458 (2000).
21) Griswold, W.: Coping with change using information transparency, Technical Report CS98-585, University of California, San Diego, Department of Computer Science and Engineering (1998).

(平成 16 年 9 月 2 日受付)
(平成 17 年 2 月 1 日採録)

片岡　欣夫（正会員）

平成 3 年大阪大学基礎工学部情報工学科卒業．平成 5 年同大学大学院修士課程修了．同年株式会社東芝入社，システム・ソフトウェア生産技術研究所配属．平成 11 年より平成 13 年まで米国ワシントン大学客員研究員．平成 15 年より東芝研究開発センター研究企画室主務．平成 17 年より同センターシステム技術ラボラトリー研究主務．ソフトウェアの生産性や品質定量化手法，ソフトウェアアーキテクチャ設計に関する研究に従事．

楠本　真二（正会員）

昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 8 年同大学講師．平成 11 年同大学助教授．平成 14 年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助教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．電子情報通信学会，IEEE，PM 学会，JFPUG 各会員．

井上　克郎（正会員）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59〜61 年ハワイ大学マノア校情報工学科助教授．平成 1 年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．工学博士．平成 14 年大阪大学情報科学研究科コンピュータサイエンス専攻教授．ソフトウェア工学の研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE, ACM 各会員．