リファクタリング箇所特定支援のためのパターン記述言語

村松裕次[†1,☆] 中川晋吾[†2] 出口博章[†3]
水野忠則[†4] 太田剛[†4] 酒井三四郎[†4]

オブジェクト指向プログラム開発において設計は非常に重要である．しかし，最初から正しい設計を行うのは非常に困難である．プログラムの設計を改善する手法の 1 つにリファクタリングがある．リファクタリングを行うことで様々な恩恵が得られるが，あまり実践されていない．その主な理由の 1 つにリファクタリング箇所の特定が困難であることがあげられる．本論文ではリファクタリング箇所特定を支援するためのパターン記述言語を提案する．特徴を的確に記述するために，クラス間の関係やメソッド内部の構造などを表す表現を用意した．ユーザ自身が記述することで，プログラムの中で，ある特徴を持つ部分を細かく指定して検出することができる．ツールを実装して評価実験を行い本パターン記述言語の有用性を確かめた．

Pattern Description Language for Identifying Refactoring Opportunities

Yuji Muramatsu,[†1,☆] Shingo Nakagawa,[†2] Hiroaki Deguchi,[†3]
Tadanori Mizuno,[†4] Tsuyoshi Ohta[†4] and Sanshiro Sakai[†4]

In object oriented programming, the design is very important. However, it is very difficult to perform the good design. Refactoring is one of the techniques which improve the program design. Refactoring is to reorganize a program without changing its function. Various benefits are obtained by refactoring. However, Refactoring is seldom practiced. It is because identifying refactoring opportunities is difficult. In this paper, the authors propose the pattern description language for identifying refactoring opportunities. The language has some methods that describe relationships between classes and inner structures of method so that characteristics can be specified clearly. Users can describe the patterns by themselves so that they can specify features of refactoring opportunities in detail. The prototype tool was implemented and the evaluation experiments were performed. They showed the usefulness of this pattern description language.

1. はじめに

オブジェクト指向プログラム開発において，設計は非常に重要である．誤った設計でコーディングされたプログラムは，可読性に乏しく，またオブジェクト指向プログラムの利点である再利用性や柔軟性も失われてしまう．そのため，正しい設計を実践するために様々な設計手法が考えられてきた．

しかし，つねに正しい設計を保つことは困難である．たとえ初期段階で正しい設計を行ったつもりでも，たび重なる仕様の変更などで最後には収拾のつかない状態になっていることも少なくない．それを改善する手法の 1 つにリファクタリングがある．

リファクタリングとは，「外部から見たときの振舞いを保ちつつ，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[1)]．リファクタリングは従来からプログラマが行っていたソースコードの整理を体系づけたものである．Fowler らはリファクタリング作業に注目し，この作業の手順を洗練し，カタログ化した[1)]．その非常に統制された方法でコードを洗練していくため，修正の際にバグの入り込む余地はほとんどない．

ところが，現状ではリファクタリングはあまり実践

†1 静岡大学大学院情報学研究科
Graduate School of Informatics, Shizuoka University
†2 静岡大学大学院理工学研究科
Graduate School of Science and Engineering, Shizuoka University
†3 八戸大学ビジネス学部
Faculty of Business, Hachinohe University
†4 静岡大学情報学部
Faculty of Infromatics, Shizuoka University
☆ 現在，株式会社ジャステック
Presently with JASTEC Co., Ltd.

されてはいない．その主な理由の 1 つに，リファクタリング箇所の特定が困難だということがあげられる．膨大なソースコードの中からリファクタリングを行えそうな箇所を特定するのは，かなりの時間と労力を必要とする．

プログラム中にプログラマが「悪い設計」と考える特徴があった場合，その部分はリファクタリングを行える可能性がある．この特徴を持った部分をプログラムから検出できれば，リファクタリングを行うべき箇所が特定できると考えられる．

リファクタリング操作を行うことに関しては様々な研究がされていて，JRefactory [2] や RefactorIT [3] などの実用的なツールが開発されてきている．しかし，どこをリファクタリングできるかが分からなければ，そのツールを有効に活用することはできない．

そこで本論文では，リファクタリング可能箇所の特定を支援するためのパターン記述言語を提案する．これはプログラムの構造を記述するための言語である．これを利用することで，より円滑にリファクタリング作業を行うことを目標とする．

2. パターン記述言語の提案

2.1 特　　長

本論文では，プログラム上に現れる構造の特徴を表現する言語を提案する．クラス間関係とメソッドの内部構造の表現を同時に扱えるパターンを検出する点に特長がある．リファクタリングは基本的に構造を変化させるものである．そこで，文字列の特徴を表現するのではなく，プログラムの構造上の特徴を表現できるパターン記述言語が必要である．

リファクタリングが行えるかどうかはプログラマの主観が入り込む[4]．ある部分が設計的に悪いかどうかは，少なからずプログラマの意見や経験などに影響されてしまう．パターン記述言語でリファクタリング可能箇所の特定を支援しようとする方式はプログラマが「悪い設計」と考える特徴を記述し検出できる点に利点がある．たとえば，カタログ化されたリファクタリング[1] を行う動機となる特徴を本言語で的確に表現できれば，検出された箇所で目的とする操作がユニークに行える．しかし，プログラム構造の特徴で表現するという限界から的確に表現できない特徴もある．その限界から，目的とするリファクタリング操作と無関係な箇所と複数のリファクタリング操作が可能な箇所が混在して検出されることになる．その場合は，対象プログラム固有の情報に基づいた特徴を用いて絞り込み，リファクタリング箇所特定の支援を行う．この方法による利点は以下の点にある．

- 自分独自の検出パターンを記述できる．
- 検出された部分の特徴が把握できているので，実際にリファクタリングが適用できるかの判断の助けになる．

2.2 文法の設計方針

提案する記述言語はプログラムの構造を表現できる．ここでいう構造とは，クラス間関係，フィールド宣言などのクラス内構成や，if 文の使用，メソッド呼び出しなどのメソッド内の基本的な処理などを組み合わせたものである．また，複数のクラス間における同名のメソッドの有無や他クラスのメソッド呼び出しなどもこれに含まれる．

プログラムの構造を表現しようとする場合，そのプログラムの記述言語と同様の構造表現ができることが望ましい．そうすることで，対象の言語の構造上の特徴を自然に表現することができ，なおかつ分かりやすいという利点がある．提案するパターン記述言語は対象言語とほぼ同等の構文規則を持つ．検出対象としては Java 言語を選んだ．Java 言語はオブジェクト指向言語であり，現在最もポピュラーな言語の 1 つである．また，静的に型が関連づけられるので，構造のみの検出であっても，型によるマッチングも可能である．

また，言語の記法も分かりやすさを重視した．“すべて” を表現する場合のワイルドカード（∗）など，すでに同等の意味の記法が他に存在する場合，その記法と同じ表現を用いることにした．

2.3 文　　法

本論文で提案するパターン記述言語の文法について述べる．なお，完全な文法は付録 A に記載する．

2.3.1 クラス（インタフェース）定義

どのようなクラスが定義されているかを表現する．「public クラスである」,「クラス名が Listener で終わる」などを表現することができる．

この文法のすべての表現は，このクラス定義をもとに構成される．また，クラス定義は複数並べて記述することができ，その場合複数のクラスの関係を考慮したマッチングが可能である．

```
public class * {}
```

このように記述した場合，任意の *public* なクラスを表現する意味になる．∗ はワイルドカードの意味で任意のクラスにマッチする表現である．

2.3.2 クラス間関係

継承などの関係を表現することができる．それぞれのクラスがどのような関係にあるかを記述できる．

```
class * extends * {}
```

このように記述すると何かを明示的に継承しているクラスにマッチする．以下で説明する変数などを利用すれば，より詳細に検出するパターンを表現できる．

2.3.3 名前表現

クラス名や型名など，名前を記述できるところには以下の記述が可能である．

- ワイルドカード（*）

任意の文字列にマッチする．以下のように書けば任意のクラスにマッチするパターンを表現できる．

```
class * {}
```

- 変数（$name, @name）

基本的にワイルドカードと同じ意味で任意の文字列にマッチする．ただし，同じ変数名が 2 つ以上記述されている場合それぞれの変数には同じ文字列がマッチする．変数は@，$で始まる 2 種類があり，これは個数表現（2.3.6 項）をした場合に影響する．

以下の例は，継承関係にあるクラスの組を検出するパターンである．

```
1: class @super {}
2: class @sub extends @super {
3:   $type *;
4:   @type *;
5: }
```

例では@super という変数を 2 カ所で使用している．この場合，1 行目のクラス名と 2 行目でスーパクラスとして指定したクラス名が同じである箇所にマッチするということを意味する．また@sub は@super とは異なるクラス名の箇所にマッチする．フィールド定義の部分の$type は他の変数とは無関係に任意の文字列にマッチするが，@type は@sub，@super とは異なった文字列にマッチする．

- 接頭語，接尾語（*^xx-yy）

変数には接頭語，接尾語の指定ができる．*^xx-yy と記述すると xx から始まる，yy で終わる文字列にマッチする．

```
1: class * {
2:   int $field;
3:   int $field^get() {}
4:   void $field^set() {}
5: }
```

このように記述すれば$field に対するアクセサメソッド相当のものを表現可能である．Java ではフィールド名が name だとしたら get メソッドは getName() のように書くのが慣例である．そのため接頭語，接尾語を指定した場合，変数マッチのときに大文字小文字の区別をなくすことにした．また，このとき@field は name という文字列にマッチしたことになる．名前表現部で指定できる表記としては，現時点では，最低限必要な機能を実現しており，正規表現などの複雑な表現方法は用意されていない．

2.3.4 フィールドとメソッドの定義

クラス定義の内部には，フィールド，メソッドの定義を書くことができる．メソッド定義には引数も含めることができる．このとき引数は指定された順番とは無関係にマッチする．つまり，順番が異なっていたり数が違っていたりしても，指定された引数がすべて定義されていればマッチする．また，メソッド定義の引数の部分に “..” を指定すると引数は無視してマッチする．以下の例では int 型のフィールドと戻り値の型が void であるメソッドを少なくとも 1 つ持ったクラスにマッチする

```
1: class * {
2:   int $field;
3:   void *(..) {}
4: }
```

2.3.5 メソッド内部の構造表現

ローカル変数定義や代入など，メソッド内部の表現が可能である．以下の表現はすべて，メソッド定義のブロック内に表記することが可能である．

- 任意の文

_stmt_は任意の文を意味する．以下のように書くことで 10 以上の文がある箇所を検出できる．

```
_stmt_<10,>;
```

- ローカル変数定義

メソッド内部で定義された変数を表現する．以下のように型と変数名を指定する．

```
$type @local;
```

- 参照

メソッド，変数などが使用されていたときにマッチする．

```
1:  @type @local;
2:  @local;
3:  @type#*();
4:  @local.*();
```

1 行目でローカル変数を定義している．2 行目ではその@local という変数が記述されている箇所にマッチする．このときそのメソッドやフィールドの参照も表現することができる．‘#’ と‘.’ を表記することでその意味を持たせる．3 行目は@type クラスの任意のインスタンスに対してメソッドを呼び出しているところにマッチする．一方，4 行目は@local という変数で

参照されるオブジェクトに対するメソッド呼び出し部分にマッチする．

- 代入

代入が行われている箇所にマッチする．以下のように表記すると，@local に代入している箇所がマッチする．

```
@local = *;
```

- return 文

以下の例は，@local を返り値としている return 文にマッチする．

```
return @local;
```

- 制御文

以下の制御構造を表す表現ができる．

`if, switch, branch, while, for, loop, block`

loop は while と for，branch は if と switch，block はこれらすべてにマッチする表現である．

以下の例は条件部に@local を参照している if 文にマッチする．ブロック内にはメソッドブロックと同様の記述ができる．その他の制御文も同様の文法で記述できる．

```
if(@local) {}
```

2.3.6 個数表現

メソッド定義やフィールド定義などには個数表現が記述でき $< m, n >$ のように表現する．この場合，m 以上 n 以下の対応するものが定義されているときにマッチする．また，数値のどちらかは省略することも可能である．この場合，省略された方の制限はなくなる．以下の例は int 型のフィールドが 3 つ以上定義されているクラスにマッチする．

```
1: class * {
2:   int *<3,>;
3: }
```

また，回数を指定している箇所に変数が使われている場合，@は異なる文字列がマッチした数，$は同じ文字列がマッチした数という制限になる．

たとえば，以下の対象プログラムからパターンを検出した場合，$local1 は同じものが 3 回以上使われている‘a’にマッチし，@local2 は‘a’，‘b’，‘c’の 3 種類にマッチする．

```
// パターン
1: * @method(..){
2:   $local1<3,>;
3:   @local2<3,>;
4: }
// 対象プログラム
1: void method() {
2:   int a, b, c;
3:   a = 0;
4:   b = a;
5:   a = c;
6: }
```

2.4 パターン記述例

前節の記述方式を用いて，プログラムの構造を表現した例を示す．いくつかのリファクタリングを行うべき悪い特徴と，それを表現するプログラムパターンの例を示す．

(1) 問合せと更新の分離

このリファクタリングが行える可能性のある特徴

- 1 つのメソッドが値を返すと同時にオブジェクトの状態を変更している

この特徴を表現しているパターンを以下に示す．

```
1 : class * {
2 :   $type $field;
3 :   * $other;
4 :   public * *($type $arg) {
5 :     $field = $arg;
6 :     return $other;
7 :   }
8 : }
```

4–7 行目で表現されているメソッドは基本的に@other というフィールドの値を得る問合せのメソッドである．これは 6 行目で表現される．しかし，それと同時に 5 行目でフィールドに引数の値を代入している．これは更新である．つまり，問合せと更新の作業を同一メソッドで行っていると表現している．

(2) パラメータへの代入の除去

このリファクタリングが行える可能性のある特徴

- 引数への代入が行われている

この特徴を表現しているパターンを以下に示す．

```
1 : class * {
2 :   * *(* @arg) { @arg = *; }
3 : }
```

2 行目で定義した引数@arg に代入を行っているメソッドを表現している．

(3) 仲介人の除去

このリファクタリングが行える可能性のある特徴

- クラスのやっていることが単純な委譲のみ

この特徴を表現しているパターンを以下に示す．

```
1: class @server {
2:   $delegate $del;
3:   public * $todel(..)<3,> {
4:     _stmt_<1>;
```

```
 5:     $del.$todel();
 6:   }
 7: }
 8: class $delegate {
 9:   public * $todel(..) {}
10: }
```

2 行目は委譲しているクラスをフィールドとして持つことを表している．そのクラスは 8–10 行目で表現している．`$todel` メソッドは委譲メソッドで@server クラスと`$delegate` クラス両方に同じ名前のメソッドで定義されていることを表す．4 行目と 5 行目では @server クラスの`$todel` メソッドが`$delegate` クラスのメソッドを呼んでいるだけであることを表している．他クラスにあるメソッドに単純に委譲するメソッドが 3 つ以上あるクラスを検出できる．

3. 実　　装

3.1 全体的な流れ

本ツールは java で実装を行った．実装したシステムは 2 つに分けることができる．ソースコードの XML 変換部とパターン検出部である．XML 変換部では，システムは対象となる Java ソースコードを XML ツリーに変換する．XML ツリーにリファクタリング箇所を検出するのに必要な情報を埋め込む必要があるため，独自に作成した．パターン検出部では，記述言語で書かれた検出パターンと XML ツリーをもとにパターンマッチングを行い，処理結果を表示する．パターン記述言語の構文木の生成には javacc☆を利用した．

3.2 パターンマッチングアルゴリズム

次のサンプルパターンと検出対象ソースコードをもとにアルゴリズムを説明する．

```
1: // サンプルパターン
2: class * {
3:   * $field;
4:   * @method(* @arg) { $field = @arg; }
5: }
```

```
1: // 検出対象となるソースコード
2: class A {
3:   int a, b;
4:   void m1(int c) { c = 0; }
5: }
6: class B {
7:   int e;
8:   void m2(int f) { e = f; }
```

☆ https://javacc.dev.java.net/

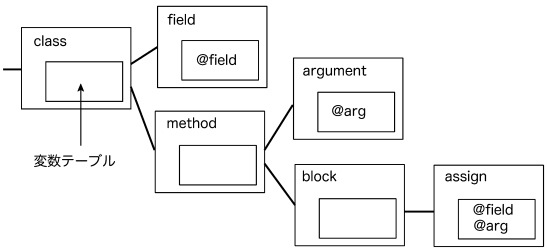

図 1 サンプルパターンの構文木
Fig. 1 Syntax tree of the sample pattern.

```
9: }
```

まず，記述言語で書かれたパターンを解析し，構文木を作る（**図 1**）．クラス，フィールド，メソッドなどを節とした木になる．それぞれの節には変数テーブルがあり，ここには`@field` などで適合した文字列を入れておく．

この構文木に対して，対象となるソースコードをクラスごとに重ね合わせるようにマッチングを行っていく．例ではまずクラス A のマッチングを行い，その後クラス B のマッチングを行う．

クラス A についてのマッチングの流れを示す．

- クラスのシグネチャについてマッチングを行う．クラス名は ‘*’ で他の指定はないので適合する．
- フィールドについてマッチングを行う．フィールドは a，b の 2 つある．変数名で`$field` と指定がある．ここでは a，b の 2 つはパターンにマッチするので変数テーブルに`$field` を ‘a’，‘b’ の文字列として登録する．
- すべてのフィールドを探索したあと，変数テーブルの情報を親のテーブルに渡す．ここではクラスの変数テーブルに`$field` の情報を渡す．これはあとで変数名での絞り込みを行う際に利用する．
- メソッドのシグネチャについてマッチングを行う．変数テーブルに`@method` を m1 として登録しておく．
- 引数のマッチングを行う．変数テーブルには`@arg` を c として登録する．
- 代入文についてマッチングを行う．`$field`，`@arg` はともに以前に出てきたものである．親の変数テーブルを調べていくと，クラスの変数テーブルに`$field` を ‘a’，‘b’ として登録してある．代入文の左辺は ‘c’ なのでこの代入文はパターンに適合しない．2 度目以降に出てきた変数名については，親の変数テーブルを見てそれに登録されていないかどうかで絞り込む．

このようにマッチングを行っていき，`$field=‘e’`

表 1 本パターン記述言語の表現力
Table 1 Expression ability of the pattern description language.

表現の程度	表現数	小計		合計
a：特徴のほとんどを表現	9			
b：特徴の一定程度を表現	24	33		
c：あいまいな表現	21		54	
d：表現できない	18			72

`@method='m2' @arg='f'` でクラス B のメソッド m2 の部分がパターンに適合するということを見つける．

4. 評価と考察

4.1 本言語の表現力

本言語の表現力を評価する目的で，悪い設計の特徴としてよく知られている Fowler が示す 72 種類のリファクタリング操作[1] について，そのリファクタリングを行う動機となる特徴を本言語を用いて記述した．その特徴をどの程度忠実に再現できたかの概要を**表 1** に示す．また，その詳細を付録 A.2 に示す．

表現の程度は，「a：特徴のほとんどを表現」，「b：特徴の一定程度を表現」，「c：あいまいな表現」，「d：表現できない」の 4 段階で表す．「a：特徴のほとんどを表現」は特徴のほとんどを表現でき，本来検出すべき箇所を検出する割合（再現率）が高く，検出された箇所で目的のリファクタリングが行える割合（適合率）も高い．「b：特徴の一定程度を表現」では，特徴のうち，ある側面は表現できるが，表現できない側面もある．この場合，再現率は高いが，関係のない箇所も検出してしまう可能性が高い．「c：あいまいな表現」は，特徴の一部分しか表現できないために，他のリファクタリングの動機となる特徴との区別ができないものである．そのため，検出された箇所は何種類かのリファクタリングが可能な箇所を含んでいる可能性が高い．また，関係のない部分も検出してしまうため，そのままではリファクタリング箇所の特定支援とはいえない程度のものである．

ただし，これらは，「リファクタリングが可能であるとされる一般的な特徴」だけを表現しようとしている．本評価実験では行っていないが，b や c に関して，メソッド名など対象プログラム固有の情報を付け加えることによって，リファクタリング箇所特定支援として利用できる可能性がある．ただし，その結果として，本来検出すべき箇所を検出できない危険度が上昇する恐れがある．

72 種のうち，なんらかのパターンで表現できた特徴は 54 種であった．「特徴のほとんどを表現」は 9 種類，「特徴の一定程度を表現」は 24 種類あった．「あいまいな表現」は 21 種類あった．この場合は検出箇所が多くなるため，対象プログラム固有の特徴を付加して絞り込まないと実用的ではない．たとえば「引数が多い」という特徴は次のように書ける．

```
* *(* *<5,>);
```

この条件だと，「メソッド呼び出しによる引数の除去」，「オブジェクトそのものの受け渡し」，「引数オブジェクトの導入」などが行える可能性があるが，検出された箇所を詳しく調べてみないと分からない．これらのように，検出された箇所に対して，どのリファクタリングを行えるかを絞れないような場合の表現である．

あいまいな表現になるほど，検出された箇所の精度は落ちる．しかし，「特徴の一定程度を表現」のパターンであれば，その箇所を検出して検出箇所を絞り込むことが可能である．

リファクタリング可能箇所の判断基準になる特徴で，本記述言語で表現できないものには以下のようなものがあった．

(1) …する必要がある
(2) …と同じ，類似の処理をしている

(1) は「メソッドがより多くの情報を必要としている」，「2 つのクラスが互いにその特性を使う必要がある」などの特徴である．メソッドに引数を追加するなどのリファクタリングを行える可能性があるが，このような場合は，それぞれ状況が違うので，プログラムに特徴的なパターンは現れないと考えられる．

(2) については，パターンを記述するという手法自体の限界である．たとえば，「同じ処理をするアルゴリズム」などはパターン記述という手法では検出することができない．

4.2 検出箇所

表 1 の「特徴のほとんどを表現」，「特徴の一定程度を表現」できた 33 種のパターンについて実際にソースコードから検出を行ったところ，12 パターンについて，マッチした箇所が見つかった．対象ソースコードは，XML 文書の変換処理を多く含む音声ブラウザ[5] で，クラス数 84，行数 30,000 行である．**表 2** にその結果を示す．

目的操作可能は，検出された場所のうち特徴で示されるリファクタリング操作が行えた箇所の数，なんらかの操作可能は，検出された場所について目的のリファクタリング以外の操作が可能であった場所の数である．表 2 の上 3 行の操作は「特徴のほとんどを表現」で表せたもの，下 9 行は「特徴の一定程度を表現」

表 2 検出実験の結果
Table 2 The result of detection tests.

リファクタリング操作	検出時間（秒）	検出箇所	目的操作可能	なんらかの操作可能
引数への代入の除去	0.5	8	5	3
フィールドのカプセル化	0.02	5	5	0
サブクラスによるタイプコードの置き換え	0.4	5	3	0
フィールドの移動	0.2	26	3	0
一時変数の分離	0.8	14	2	7
委譲の隠蔽	0.5	13	1	0
引数の除去	1.0	12	0	7
自己カプセル化フィールド	0.2	11	2	0
クラスのインライン化	0.05	7	3	0
クラスによるタイプコードの置き換え	0.4	7	2	0
null オブジェクトの導入	0.5	2	0	2
制御フラグの削除	0.3	1	0	0

で表せたものである．

本ツールを用いることにより，人手でエディタの検索機能などを利用して 30 分以上かけて探した箇所が 1 秒以内で検出された．リファクタリングが可能であるとされる一般的な特徴を表現したパターンはライブラリとしてツールに装備する予定で，作成済みである．本実験ではこの記述済みのパターンを利用した．

「特徴のほとんどを表現」で表されたパターンについては，目的のリファクタリングの適用が可能な箇所が 72%,「特徴の一定程度を表現」で表されたパターンについては目的のリファクタリングを行える箇所は 13%であった．表 1 の表現の程度の分類によって検出精度に差が出る傾向が確認できた．

「特徴の一定程度を表現」で表されたパターンでは検出箇所が多くなった．これはパターンが細かいところまで指定できず，目的のリファクタリングとは関係ない箇所も検出してしまっているからである．たとえば，一時変数の分離では，一時変数に複数回代入されている箇所を探した．そのときプログラムの制御フローを無視してマッチング処理をしているため，条件分岐のそれぞれで代入をしている箇所にも適合してしまった．

一方，検出箇所において，目的以外の何らかのリファクタリング操作が可能な箇所があった．たとえば，異なるリファクタリング操作の動機となる特徴をパターンとして正確に区別して表現することができない場合，同じパターンとなってしまう．その場合，そのパターンによって検出される箇所には，複数のリファクタリング操作が可能な箇所が混ざって検出されることになる．このとき，目的とするリファクタリング操作以外の操作が可能であり，一定の効果が認められた．

表 2 の上 3 行の操作が行える特徴に基づいて，3 人の大学生に，同じソースコードを対象にリファクタリング可能な箇所を手作業で検出させた．その結果，3 人の検出した箇所の和集合は表 2 の検出箇所と一致した．

以上の結果より，リファクタリングが可能かどうかを調べる際に，対象となる箇所を絞り込むのには十分に役に立つといえる．

4.3 リファクタリングの適用例

表 2 の「サブクラスによるタイプコードの置き換え」では以下のようなクラスを検出することができた．

```
class BranchStatement extends SimpleNode {
  public static final int IF = 0;
  public static final int SWITCH = 1;
  ..
  public static final int BLOCK = 6;
  private int type;
  public void setType(int type) {
    this.type = type;
  }
  public List getBranchStatement() {
    List result = new ArrayList();
    switch(type) {
    case IF:
      result.add("if");
      ..
    case SWITCH:
      ..
    case BLOCK:
      ..
    }
    return result;
  }
}
```

フィールドのタイプによって処理を変えている部分が検出された．次に，以下のパターンを用いて type を設定している箇所を検出した．

```
class * {
  * *(..) {
    BranchStatement#setType();
  }
}
```

その結果，以下のように BranchStatement オブジェクトを生成したあと，すぐに type を設定している箇所が検出された．

```
BranchStatement node
                  = new BranchStatement();
node.setType(BranchStatement.IF);
```

そこで，7 つのフィールドそれぞれをサブクラスにすることで，BranchStatment クラス内の switch 文を消すことができた．

```
class IfBranchStatement
              extends BranchStatement {
  public List getBranchStatement() {
    List result = new ArrayList();
    result.add("if");
    return result;
  }
}
class SwitchBranchStatement
              extends BranchStatement {
  ..
}
abstract class BranchStatement {
  public abstract List getBranchStatement();
}
```

次に，BranchStatement を生成している箇所を以下のように修正した．このように検出された場所に目的のリファクタリングをほどこすことができた．

```
BranchStatement node
              = new IfBranchStatement();
```

5. 関連研究

リファクタリング箇所検出に関して，いろいろな側面からの研究が行われている．

肥後らはコードクローン解析に基づきリファクタリングを試みている[6]．ソースコードを字句解析し，トークン列にし，それを変数名の違いを吸収するなどの変換ルールを用いて変換したあと，一定の長さ以上一致している部分をクローンペアとして検出している．本記述言語では，記述したパターンで類似した構造上の特徴を有した箇所を検出するため，検出される箇所は類似した構造上の特徴を有している．そのため，検出された箇所にクローンコードが含まれる可能性を期待できる．また，あるコードについて，そのコードを抽象化したパターンとして記述することで，そのコードと似ている箇所を検出することもできる．このように，結果的にクローンコードを検出できる可能性はあるが，直接的にクローンペアを探すことはできない．

Kataoka らは不変式を利用してリファクタリング箇所の検出を試みている[7]．プログラムを実行して得られる不変式を利用してリファクタリング箇所を特定している．動的な結果を利用するため，本論文の方式とは違う箇所を検出することができる．

これら 2 つは，検出対象を限定してリファクタリング可能箇所の検出を試みている．本記述言語ではクラス間の関係，メソッドの内部の構造に注目し，これらより広い視点からの検出が可能である．

秦野らは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を提案している[8]．ソースコードから算出したメトリクスを利用し，その中から複雑度の高い箇所にその複雑度を低くするリファクタリングを施す．それぞれのリファクタリング操作に対して，どの複雑度を下げるかということが分かる表を作っている．プログラムから算出したメトリクスと，この表を照らし合わせることでリファクタリングの箇所を特定する．本記述言語は，パターンを記述し，具体的に該当コードを検出し，それらを列挙することでリファクタリング可能かどうかを判断することができる．パターンに適合した部分のみを抽出することで，該当箇所のコードの状況の把握を簡単にできる．

Grant らはソースコードから問題のある箇所を TXL[9] を利用して検出し，それを XML タグで囲む手法を提案している[10]．その XML のブロックを分かりやすい形でグラフィカルに出力することができる．しかし，この方法ではクラス間の関係など大域的な関係をもとにした検出ができない．

また，Jahnke らはデザインパターンを用いた設計の悪い部分を特定するために，GFRNs[11] というグラフィカルな言語を利用することを提案している[12]．この言語はあいまいな知識を定義し，それを分析することができる．しかし，この言語ではメソッド内部の構造に関する特徴を表現できない．

本言語では，リファクタリング箇所を，クラス間の関係，メソッド内部の構造を同時に表現することで，より幅広い検出を可能にしている．

ソースコードを解析し，様々な問合せを実現できるシステム Sapid[13] がある．本システムの実装を支援できる可能性があるが，リファクタリングのための問合せをその場でプログラミングできるほど容易ではなく，何らかのパターン記述言語を必要とする．一方，このようなパターン記述言語の提案[14] もある．本提案手法に比べて，きめ細かな特徴を表現でき，柔軟性もあるが，個数表現ができないか，もしくは間接的にしかできず，リファクタリング箇所の特定という用途にとって，不十分である．

6. おわりに

現在，ソフトウェアは巨大化の一途をたどり，他人の書いたコードを修正する必要性が増大している．こうした中，リファクタリングとそれをサポートする手法はますます重要になっていく．本論文では，リファクタリングを行うべき箇所を検出する方法としてパターン記述言語を提案した．これによって自分の検出したい部分を短時間で絞り込むことができるようになった．

しかし，課題も多い．今回実装した検出システムは

まだ試作レベルであり，ユーザインタフェースに関してはまだ不便なところがある．これらは検出された箇所を評価するときの効率に大きく関わってくるので，ツールとしての完成度を上げる必要がある．

参 考 文 献

1) Fowler, M., Beck, K., Brant, J., Opedyke, W. and Roberts, D.: Refactoring: Improving the Design of Existing Code, Object Technology International, Inc..
児玉公信，友野晶夫，平澤　章，梅澤真史（訳）：リファクタリングプログラミングの体質改善テクニック，ピアソン・エデュケーション (2000).
2) Seguin, C.: JRefactory Homepage (2004). http://jrefactory.sourceforge.net/
3) Aqris: RefacotorIT Fact Sheet (2004). http://www.refactorit.com/
4) van Emden, E. and Moonen, L.: Java Quality Assurance by Detecting Code Smells, *Proc. 9th Working Conference on Reverse Engineering*, IEEE Computer Society Press (2002).
5) 滝沢達也，酒井三四郎：情報教育における視覚障害者向け音声ブラウザ，教育システム情報学会第 27 回全国大会論文集，pp.373–374 (2002).
6) 肥後芳樹，神谷年洋，楠本真二，井上克郎：コードクローン解析に基づくリファクタリングの試み，情報処理学会論文誌，Vol.45, No.5, pp1357–1366 (2004).
7) Kataoka, Y., Ernst, M.D., Griswold, W.G. and Notkin, D.: Automated Support for Program Refactoring using Invariants, *Proc. Internatial Conference on Software Maintenance* (*ISCM'01*) (2001).
8) 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).
9) Cordy, J.R. and Carmichael, I.H.: The TXL Programming Language Syntax and Informal Semantics Version 7, External Techical Report, pp.93–355 (1993).
10) Grant, S. and Cordy, J.R.: Automated Code Smell Detection and Refactoring by Source Transformation, *WCRE Workshop on REFactoring: Achievements, Challanges, Effects*, Victoria (Nov. 2003).
11) Jahnke, J. and Zundorf, A.: Generic Fuzzy Reasoning Nets as a basis for reverse engineering relational database applications, *Proc. ESEC'97* (1997).
12) Jahnke, J. and Zundorf, A.: Rewriting poor Design Patterns by good Design Patterns, *ESEC/FSE'97* (1997).
13) 福安直樹，山本晋一郎，阿草清滋：細粒度ソフトウェア・リポジトリに基づいた CASE ツール・プラットフォーム Sapid，情報処理学会論文誌，Vol.39, No.6, pp.1990–1998 (1998).
14) Paul, S. and Prakash, A.: A Framework for Source Code Analysis using Program Patterns, *IEEE Trans. Softw. Eng.*, Vol.20, No.6, pp.463–475 (1994).

付　　録

A.1 パターン記述言語文法

```
Program ::= { TypeDecl }
TypeDecl ::= ClassDecl | InterfaceDecl
ClassDecl ::= Prefixes ‘class’
   ClassSignature ClassBody
InterfaceDecl ::= Prefixes ‘interface’
   InterfaceSignature InterfaceBody
Prefixes ::= { Prefix }
Prefix ::= ‘public’ | ‘abstract’ | ‘final’
   | ‘private’ | ‘protected’ | ‘static’
ClassSignature ::= Name [ Inheritance ]
   [ implementation ]
InterfaceSignature ::= Name
   [ InheritanceList ]
Inheritance ::= ‘extends’ Name
InheritanceList ::= ‘extends’ NameList
Implementation ::= ‘implements’ NameList
ClassBody ::= ‘{’ { ClassDetail } ‘}’
ClassDetail ::= ClassDecl | FieldDecl
   | MethodDecl | ConstructorDecl
FieldDecl ::= Prefixes TypeName
   Name Times ‘;’
ConstructorDecl ::= Prefixes
   ConstructorSignature Times MethodBody
MethodDecl ::= Prefixes MethodSignature
   Times MethodBody
MethodSignature ::= TypeName Name
   ‘(’ [ Params ] ‘)’
ConstructorSignature ::= Name
   ‘(’ [ Params ] ‘)’
MethodBody ::= ‘;’ | ‘{’ Statements ‘}’
Statements ::= { Statement }
Statement ::= BasicStatement
   | BlockStatement
BlockStatement ::= AnyStatement
   | IfStatement | WhileStatement
   | ReturnStatement | Block
```

```
AnyStatement ::= '_stmt_'
ReturnStatement ::= 'return' [ Name ] ';'
IfStatement ::= 'if' '(' [ Name ] ')'
   Times Block
WhileStatement ::= 'while' '(' [ Name ] ')'
   Times Block
Block ::= '{' Statements '}'
BasicStatement ::= VarDecl | Expression
Expression ::= Reference [ '=' Reference ]
   Times ';'
Reference ::= MethodReference
   | VarReference
MethodReference
   ::= Name '(' [ NameList ] ')'
VarReference ::= Name
VarDecl ::= TypeName Name Times ';'
Params ::= '..' | TypeNameList
TypeNameList ::= TypeAndName
   { ',' TypeAndName }
TypeAndName ::= TypeName Name Times
NameList ::= Name { ',' Name }
TypeName ::= Name
Name ::= ( '*' | ID | VAR ) Addition
Addition ::= [ '^' ID ] [ '-' ID ]
Times ::= ['<'(NUMBER['','[NUMBER ]]|'',
   NUMBER )'>']
VAR ::= ('@'|'$') ID
ID ::= 英字 { 英字 | 数字 }
NUMBER ::=  { 数字 }
```

ただし，[ ] という表記は省略可を表す

A.2 リファクタリング操作に対するパターン記述

個々のリファクタリング操作に対しての表現を以下のように分類して，**表 3**〜**表 9** に示す．

◎…特徴のほとんどを表現
○…特徴の一定程度を表現
△…あいまいな表現
×…表現できない

表 3 メソッドの構成
Table 3 Composing methods.

リファクタリング操作	表現評価
メソッドの抽出	○
メソッドのインライン化	◎
一時変数のインライン化	○
問合せによる一時変数の置き換え	◎
説明用変数の導入	○
一時変数の分離	○
パラメータへの代入の除去	◎
メソッドオブジェクトによるメソッドの置き換え	○
アルゴリズムの取り換え	×

表 4 オブジェクト間での特性の移動
Table 4 Moving features between objects.

リファクタリング操作	表現評価
メソッドの移動	○
フィールドの移動	○
クラスの抽出	△
クラスのインライン化	○
委譲の隠蔽	○
仲介人の除去	○
外部メソッドの導入	×
局所的拡張の導入	×

表 5 データの再編成
Table 5 Organizing data.

リファクタリング操作	表現評価
自己カプセル化フィールド	○
オブジェクトによるデータ値の置き換え	△
値から参照への変更	×
参照から値への変更	×
オブジェクトによる配列の置き換え	△
観察されるデータの複製	×
単方向関連の双方向への変更	×
双方向関連の単方向への変更	○
シンボリック定数によるマジックナンバーの置き換え	△
フィールドのカプセル化	◎
コレクションのカプセル化	△
データクラスによるレコードの置き換え	△
クラスによるタイプコードの置き換え	○
サブクラスによるタイプコードの置き換え	◎
State/Strategy によるタイプコードの置き換え	◎
フィールドによるサブクラスの置き換え	△

表 6 条件記述の単純化
Table 6 Simplifying conditional expression.

リファクタリング操作	表現評価
条件記述の分解	○
条件記述の統合	×
重複した条件記述の断片の統合	△
制御フラグの削除	○
ガード節による入れ子条件記述の置き換え	△
ポリモーフィズムによる条件記述の置き換え	◎
ヌルオブジェクトの導入	○
表明の導入	×

表 7 メソッド呼び出しの単純化
Table 7 Making method calls simpler.

リファクタリング操作	表現評価
メソッド名の変更	○
引数の追加	×
引数の削除	○
問合せと更新の分離	◎
メソッドのパラメータ化	×
明示的なメソッド群による引数の置き換え	×
オブジェクトそのものの受渡し	△
メソッドによる引数の置き換え	△
引数オブジェクトの導入	△
set メソッドの削除	○
メソッドの隠蔽	◎
Factory Method によるコンストラクタの置き換え	○
ダウンキャストのカプセル化	○
例外によるエラーコードの置き換え	×
条件判定による例外の置き換え	×

表 8 継承の取り扱い
Table 8 Dealing with generalization.

リファクタリング操作	表現評価
フィールドの引き上げ	○
メソッドの引き上げ	○
コンストラクタ本体の引き上げ	△
メソッドの引き下げ	△
フィールドの引き下げ	△
サブクラスの抽出	×
スーパクラスの抽出	△
インタフェースの抽出	○
階層の平坦化	×
Template Method の形成	×
委譲による継承の置き換え	△
継承による委譲の置き換え	△

表 9 大きなリファクタリング
Table 9 Big refactorings.

リファクタリング操作	表現評価
継承の分割	×
手続的な設計からオブジェクトへの変更	△
プレゼンテーションとドメインの分離	×
階層の抽出	△

(平成 17 年 3 月 10 日受付)
(平成 17 年 10 月 11 日採録)

村松 裕次

2003 年静岡大学情報科学科卒業．2005 年静岡大学大学院情報学研究科情報学専攻修了．在学中はソフトウェア工学，特にオブジェクト指向分析・設計の研究に従事．同年（株）ジャステック入社．ネットワーク監視ソフトウェア開発に従事．

中川 晋吾（学生会員）

1978 年生．2001 年静岡大学情報学部情報科学科卒業．2002 年静岡大学大学院情報学研究科修士課程修了．現在，同大学院理工学研究科博士後期課程に在学中．ソフトウェア・コンポーネント技術に興味を持つ．日本ソフトウェア科学会学生会員．

出口 博章（正会員）

1960 年和歌山大学学芸学部数学科卒業．1996 年筑波大学経営・政策科学研究科修士課程修了．三菱電機（株）を経て，1997 年より八戸大学商学部（現ビジネス学部）教授．情報システム開発，遠隔教育，協調学習等に興味を持つ．共著『コンピュータ概論』（共立出版），共著『情報管理概論』（共立出版），日本セキュリティ・マネージメント学会，オフィス・オートメーション学会，経営情報学会，教育システム情報学会各会員．

水野 忠則（フェロー）

1945 年生．1968 年名古屋工業大学経営工学科卒業．同年三菱電機（株）入社．1993 年静岡大学工学部情報知識工学科教授，現在，情報学部情報科学科教授．工学博士．情報ネットワーク，モバイルコンピューティング，放送コンピューティングに関する研究に従事．著訳書としては『コンピュータネットワーク概論』（日経 BP），『モダンオペレーティングシステム』（ピアソン・エデュケーション）等がある．電子情報通信学会，IEEE，ACM 各会員．当会フェロー，監事．

太田　剛（正会員）

1964 年生．1989 年静岡大学大学院電子科学研究科電子応用工学専攻中退．同年静岡大学工学部情報知識工学科助手．1998 年同大学情報学部情報科学科講師．2000 年同大学同学部助教授．博士（情報科学）．ソフトウェア開発環境の研究に従事．電子情報通信学会，教育システム情報学会各会員．

酒井三四郎（正会員）

1956 年生．1984 年静岡大学大学院電子科学研究科博士後期課程（電子応用工学専攻）修了．学習院大学，新潟産業大学，静岡大学工学部を経て，1998 年静岡大学情報学部助教授．現在，同学部教授．工学博士．ソフトウェア開発支援環境，プログラミング教育支援環境，遠隔学習，協調学習に関する研究・開発に従事．電子情報通信学会，教育システム情報学会，日本 e-Learning 学会各会員．