情報処理学会論文誌 Vol. 50 No. 2 860–871 (Feb. 2009)

シーケンシャルパターンマイニングを用いたコーディングパターン抽出

石尾 隆[†1] 伊達浩典[†1]
三宅達也[†1] 井上克郎[†1]

ソフトウェア開発において，何らかの理由でモジュール化が困難な機能は，複数のモジュールに分散する定型的なコード，すなわちコーディングパターンによって実装され，保守性を悪化させる要因となっている．本研究では，シーケンシャルパターンマイニング手法を用いて，コーディングパターンをソースコードから自動抽出し，開発者によるコーディングパターンの保守を支援する手法を提案する．具体的には，Java ソースコードの各メソッドに対して，パターンマイニングのための正規化を行い，シーケンシャルパターンマイニングのアルゴリズムの 1 つである PrefixSpan を適用する．評価実験として，6 つの Java プログラムに対して提案手法を適用し，得られた頻出パターンを調査した．その結果，モジュール化が困難なアプリケーション固有の機能の実装や，Java プログラミングにおけるイディオムなど，開発者にとって有用なコーディングパターンを抽出できることを確認した．

Coding Pattern Extraction Using a Sequential Pattern Mining Approach

Takashi Ishio,[†1] Hironori Date,[†1] Tatsuya Miyake[†1] and Katsuro Inoue[†1]

To develop software, developers use idiomatic code fragments, or coding patterns, to implement a particular kind of concerns that are hard to modularize in the software. In this paper, we propose an application of a sequential pattern mining approach to extract coding patterns from source code. Our approach first normalizes Java source code to a sequence database and then applies PrefixSpan, a pattern mining algorithm, to the database. We have implemented the approach as a tool and applied the tool to six Java programs. The resultant patterns help developers to understand and maintain the programs since the patterns include application-specific crosscutting implementation and Java implementation idioms.

1. はじめに

近年，多くのソフトウェア開発にオブジェクト指向プログラミングが使用されている．オブジェクト指向プログラミングの利点として，継承や多態性など，モジュール化された部品を活用するための機構がある．しかし，ソフトウェアのすべての機能を完全にモジュール化することはできず，たとえば，ロギングや同期処理といった機能は，横断的関心事と呼ばれ，複数のモジュールに分散した定型的なコードとして実装されることが知られている[16),20)]．

複数のモジュールに分散配置されるコードは，元となるソースコード片を開発者が複製し，配置先の状況に応じて適宜改変を加えるという方式で作成されることが多く，一群の定型的なコード片，すなわち**コーディングパターン**を構成する．コーディングパターンに属するコード片は互いに類似しており，また，多くは同一の機能を実現している．そのため，コード片の 1 つを変更する場合，開発者は，同一のパターンに属する他のコード片に対しても一貫した変更を適用するべきか，検討する必要がある[3),8)]．

本研究では，コーディングパターンを，メソッド呼び出し要素と，それに付随する制御構造要素（条件分岐と繰返し文）の定型的な列ととらえ，コーディングパターンをソースコードから自動的に抽出するパターンマイニング手法を提案する．具体的には，適用対象として選択した Java 言語のためのソースコード正規化ルールを用意し，Java の各メソッドを，特徴列へと変換する．その結果得られた特徴列データベースに，シーケンシャルパターンマイニングのアルゴリズムの 1 つである PrefixSpan[25)] を適用し，頻出する部分列をコーディングパターンとして抽出する．シーケンシャルパターンマイニングは，マイニング対象である要素の順番が同じであれば，無関係な要素がいくつ間に追加されても，抽出されるパターンは影響を受けない．横断的関心事のように複数のモジュールに分散したコードは，しばしば他の機能に属するコードと混ざり合うことが知られている[16)]が，そのようなコードからもパターンの抽出が可能である．

本手法で抽出されるコーディングパターンが横断的関心事などの保守に有用であるかを調査するため，提案手法をツールとして実装し，6 つの Java プログラムに対して適用した．頻出する 55 のパターンを手作業で調査した結果，ロギングのパターンをはじめとする，モ

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

ジュール化が困難なアプリケーションの機能をコーディングパターンとして抽出していることを確認した．

以降，2 章では，研究の背景と，本研究で使用したパターンマイニングのアルゴリズムについて述べる．3 章ではコーディングパターン抽出法を述べ，4 章では 6 つの Java プログラムへの適用結果を述べ，5 章では，ケーススタディの結果について考察する．6 章で関連研究について述べ，7 章でまとめと今後の課題を述べる．

2. コーディングパターン

ソフトウェアの機能をモジュールとして分割，実装することは，保守性や拡張性を向上させるために重要である．しかし，ソフトウェアのすべての機能を完全にモジュール化することはできず，そのような機能は，複数のモジュールに分散した定型的なコードとして実装される[20]．本研究では，そのように分散して実装される定型的なコードを，コーディングパターンと呼ぶ．

図 1 は，図形エディタ JHotDraw 5.4b1 における，様々な編集操作を「元に戻す」ことを可能とするための実装の一部である．編集操作ごとに実行している処理は異なるが，下線で示されるメソッド呼び出し列が共通している．このようなメソッド呼び出しのパターンを知ることは，どのように「元に戻す」仕組みが実装されているかを理解するために，また，新たな編集操作を実装するときに有用である．

定型的なメソッド呼び出し列は，しばしば，特定の制御構造をともなった記述となる．たとえば，サーバソフトウェアである Apache Tomcat には，デバッグ用のメッセージを出力する `debug` メソッドの呼び出しが多数記述されている．各呼び出しには，メッセージを記録するかどうかを判定する `isDebugEnabled` メソッドの呼び出しと `if` 文による条件分岐が付属しており，条件分岐はメッセージ出力処理の重要な構成要素となっている．そこで，本研究では，コーディングパターンをメソッド呼び出しとそれに付随する制御構造要素の列としてとらえる．

既存の定型的なコードに基づいて新たなコードを記述するとき，開発者は，しばしば既存のコード片を複製し，改変を加えるという形式での記述を行う[17]．このようなソースコードの複製は，互いに類似したコード，すなわちコードクローン[3),15)] の一種であると考えられる．しかし，コードクローン検出ツール，たとえば CCFinder[15)] は，図 1 の下線で示されたメソッド呼び出しのように，他の機能の一部として埋め込まれた短いコード片を発見するには不向きであり，コードクローン検出手法のみでは横断的関心事の実装を特定することが困難であると指摘されている[6]．

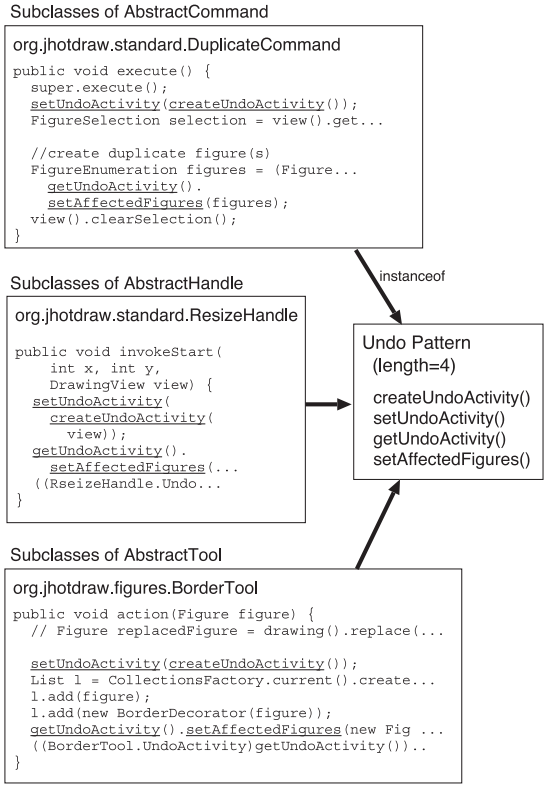

図 1　JHotDraw 5.4b1 の編集内容を「元に戻す」実装パターン
Fig. 1　Undo command pattern in JHotDraw 5.4b1.

2.1 アスペクトマイニング

本研究で抽出することを目標としているコーディングパターンのように，モジュール化することが困難な機能，すなわち横断的関心事を実装するコードを発見することに特化された手法は，アスペクトマイニングと呼ばれる．

従来のアスペクトマイニング手法は，いずれも，横断的関心事の典型的な実装法に関する経験的な指標を用いて，横断的関心事の候補を探索する．Marin らは，1 つの横断的関心

事，たとえばロギング処理を実装する多数のコード片が，`Logger.log()` のような特定のメソッドを呼び出すことから，被呼び出し回数が多いメソッドを横断的関心事の候補とする手法を提案した[21]．また，Breu らは，あるメソッド呼び出しが単独ではなく，他のメソッドと一対になって初めて意味を持つものかを調べる手法を提案している[5]．一方，Krinke は，あるメソッドの呼び出しが，必ず他のメソッドの先頭あるいは終端に配置されている，という制御フロー上の特徴を使用することを提案している[19]．

本研究で着目するコーディングパターンは，Marin らの着眼点と同様，類似したコードが多数書かれていることを手がかりとするが，メソッド呼び出し列を探索し，またそれに付随する制御構造要素をパターンの一部として含める点で，従来手法とは異なる．

2.2　シーケンシャルパターンマイニング

本研究では，シーケンシャルパターンマイニングを，コーディングパターンの抽出に使用する．シーケンシャルパターンマイニングとは，与えられた配列データベースから頻出する部分列をパターンとして抽出する手法である[2]．頻出する部分列の個々の出現は，順序が同一でなければならないが，不連続なものでよい．

本研究では，シーケンシャルパターンマイニングのアルゴリズムの 1 つである PrefixSpan を使用する．このアルゴリズムは，入力として配列データベース S と最小出現回数 min_sup をとり，以下の手順でシーケンシャルパターンの集合を抽出する[25]．

まず最初に，配列データベースに出現する各要素の出現回数（その要素を含む配列の数）を数え上げ，最小出現回数 min_sup 以上の要素を，長さ 1 のパターンとする．たとえば，$S = \{\langle abcd\rangle, \langle aeade\rangle, \langle dafb\rangle, \langle acd\rangle\}$，$min\_sup = 2$ が与えられたとき，4 つの長さ 1 のパターン $\langle a\rangle : 4$，$\langle b\rangle : 2$，$\langle c\rangle : 2$，$\langle d\rangle : 3$ が得られる．ここで，各パターンは，パターンの要素列 $pattern$ と出現回数 $support$ を用いて $\langle pattern\rangle : support$ と表記している．この例では，ただ 1 つの配列にしか出現しない $\langle e\rangle$ と $\langle f\rangle$ がパターンから除外されている．

PrefixSpan は，続いて，長さ k のパターンから長さ $k+1$ のパターンを構築する手順を，新たなパターンが抽出されなくなるまで繰り返し実行する．まず長さ k のパターンに対し，**射影データベース**（projected database）を作成する．これは，配列データベース S の各配列に対し，パターンの最初の出現以降の要素列だけを取り出す操作である．たとえば，パターン $\langle a\rangle$ に関する S の射影データベースは，$\{\langle bcd\rangle, \langle eade\rangle, \langle fb\rangle, \langle cd\rangle\}$ という 4 つの配列となる．次に，得られた射影データベース中での各要素の出現回数を数え上げ，min_sup 個以上の配列に出現した要素を用いて長さ $k+1$ のパターンを構築する．a に関する射影データベースでは，b，c が 2 つの配列に，d が 3 つの配列に出現していることから，$\langle ab\rangle : 2$，$\langle ac\rangle : 2$，$\langle ad\rangle : 3$ というパターンが得られる．S から抽出される長さ k のパターン（$prefix$），それに関する射影，そして新たに得られる長さ $k+1$ のパターンを，**表 1** に示す．記号 ϕ は，射影の結果が空であることを意味している．

表 1　PrefixSpan アルゴリズムが作成する射影データベース
Table 1　Projected databases created by PrefixSpan.

$prefix$	$prefix$-Projected Database	Patterns
	$\langle abcd\rangle, \langle aeade\rangle, \langle dafb\rangle, \langle acd\rangle$	$\langle a\rangle : 4, \langle b\rangle : 2, \langle c\rangle : 2, \langle d\rangle : 3$
$\langle a\rangle$	$\langle bcd\rangle, \langle eade\rangle, \langle fb\rangle, \langle cd\rangle$	$\langle ab\rangle : 2, \langle ac\rangle : 2, \langle ad\rangle : 3$
$\langle b\rangle$	$\langle cd\rangle$	
$\langle c\rangle$	$\langle d\rangle, \langle d\rangle$	$\langle cd\rangle : 2$
$\langle d\rangle$	$\langle e\rangle, \langle afb\rangle$	
$\langle ab\rangle$	$\langle cd\rangle$	
$\langle ac\rangle$	$\langle d\rangle, \langle d\rangle$	$\langle acd\rangle : 2$
$\langle ad\rangle$	$\langle e\rangle$	
$\langle cd\rangle$	ϕ	
$\langle acd\rangle$	ϕ	

3.　コーディングパターンマイニング

本研究では，PrefixSpan アルゴリズムを用いて，コーディングパターンを抽出する手法を提案する．本研究で抽出するコーディングパターンは，以下のように定義される．

- コーディングパターンは，メソッド呼び出し要素および制御構造要素の順序付きの列である．制御構造要素とは，条件分岐を意味する IF，ELSE，END-IF 要素と，繰返し文を意味する LOOP，END-LOOP 要素である．
- コーディングパターンに対応する具体的なコード片，すなわちインスタンスは，ソースコード上では不連続でもよい．

メソッド呼び出し単独ではなく，それに関連した制御構造も同時に抽出することで，たとえば特定の条件下でのみ実行されるロギングのパターンなどにも対応できる．また，ソースコード上では不連続な要素をパターンとして認識することで，新たなコードの追加による影響を受けないという特徴がある．

提案手法は，Java ソースコードの正規化，PrefixSpan の適用，得られたパターンのフィルタリング，そしてグループ化という 4 つのステップからなる．提案手法は，入力として，Java プログラムのソースコードと，PrefixSpan の引数である min_sup，そして最終的に提示するパターンの最小の要素数を指定する min_len を受け取る．min_len は，PrefixSpan

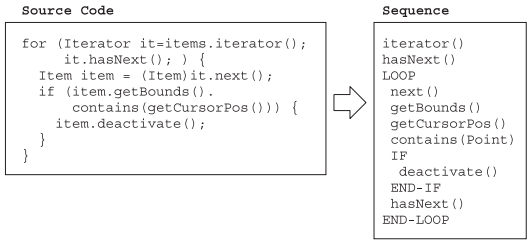

図 2　ソースコードの正規化例
Fig. 2　A sequence extracted from source code.

```
Statement: if (<cond>) <then> else <else>;
Sequence:  <cond>, IF, <then>, ELSE, <else>, END-IF

Statement: for (<init>; <cond>; <inc>) <body>;
Sequence:  <init>, <cond>, LOOP, <body>,
           <inc>, <cond>, END-LOOP

Statement: while (<cond>) <body>;
Sequence:  <cond>, LOOP, <body>, <cond>, END-LOOP
```

図 3　制御文の正規化ルール
Fig. 3　Normalization of control statements.

によって抽出される膨大な数の短いパターンを取り除くための閾値である．

正規化のステップでは，Java プログラムをメソッド単位に分解し，各メソッドのコードをそれぞれ，以下に述べるルールに従って，メソッド呼び出し要素と制御構造要素からなる配列へと変換する．要素の順序は，シーケンシャルパターンマイニングの結果に影響するため，本研究では，制御構造による呼び出し順序を正規化ルールに反映した．詳しい正規化ルールを述べる前に，まず，**図 2** に正規化の例を示す．図のコード片を実行すると，まず最初に `for` 文の初期化子で `iterator` メソッドが呼び出される．続いて，ループ本体を実行するかどうかを判定するために `hasNext` が呼び出され，ループ本体の実行に進む．ループの本体では，まず `next` を呼び出し，続く一連の処理を実行する．そして，ループ本体の末尾において，ループの継続を判定するために再度 `hasNext` を呼び出し，継続するのであればループ本体の先頭，すなわち `next` 呼び出しに戻る．図 2 の右側に示す要素列の順序は，このような制御フローを反映したものとなっている．

メソッド呼び出しの正規化．ソースコード中のメソッド呼び出し式は，メソッド呼び出し要素へと置き換えられる．メソッド呼び出し要素は，メソッドのシグネチャからクラス名を除いたもの，すなわち戻り値の型名，メソッド名，引数の型名の列である．クラス名を無視しているのは，動的束縛における同名のメソッドを同一の呼び出し要素として扱うためである．2 つ以上のメソッドが 1 つの式で呼び出される場合は，Java 言語仕様で定義された式の評価順序に従ってメソッド呼び出し要素を並べる．同一順位にある，あるいは順序関係が未定義であるようなメソッド呼び出しの順序は，ソースコード上の出現順に従って解決する．なお，図 2 の例では，誌面の都合上，戻り値の型名の表記を省略している．

条件分岐の正規化．次に，`if` 文は，**図 3** の 1 番目に示す方式で，IF，ELSE，END-IF という 3 つの要素に変換される．条件式中のメソッド呼び出しは IF 要素の直前に，IF による条件分岐の結果実行される文は，IF と ELSE の間，ELSE と END-IF の間にそれぞれ配置される．

繰返し処理の正規化．最後に，`for` 文と `while` 文は，図 3 の 2 番目および 3 番目に示す方式で，LOOP，END-LOOP という 2 つの要素に変換される．LOOP 要素は繰り返し実行される処理の先頭を，END-LOOP 要素は繰返し処理の末尾，繰返しが継続する場合は LOOP 要素に制御を戻す位置を示す．たとえば `while` 文の場合は，LOOP 要素の前と，繰り返し実行される処理の末尾，END-LOOP 要素の直前に，条件式に書かれたメソッド呼び出しが配置される．`do-while` 文の場合は，1 回目は無条件に処理を実行するため，`while` の最初の条件式の評価を取り除いた場合に等しい．

上記の正規化ルールは，`switch` 文や `try/catch`，`synchronized` には対応していない．`switch` 文は，条件式とブロックの内容をそのまま，`case` 文や `break` 文を無視して並べた形式へと変換される．他の構文も，同様にブロックの内容がそのまま展開される．また，ループ中に含まれる `break` 文，`continue` 文や `return` 文によってループの処理が中断される可能性は無視している．

正規化が完了すると，Java プログラムの各メソッドがそれぞれ 1 つの配列となった配列データベースが得られる．これに対して，PrefixSpan アルゴリズムを適用し，得られたパターン集合から，要素数が min_len に満たないパターンを取り除く．

本手法では，制御構造をパターンの要素として導入したが，メソッド呼び出しを持たない，制御構造だけからなるパターンは開発者にとって意味を持たない．そのため，得られたパターン集合に対して，以下の 2 つのルールを適用しフィルタリングを行う．

- 制御構造要素がパターンの要素数の 70%を超えたとき，そのパターンを取り除く．閾値は，経験的に定めたものである．

- 制御構造要素は，必ず IF と END-IF，LOOP と END-LOOP というように一対で生成されるため，パターンが一方だけを含み，対応する要素を含まないとき，そのパターンを取り除く．

フィルタリングを適用した後，パターンのグループ化を行う．これは，あるパターンが存在するとき，そのパターンと少なくとも同数のインスタンスを持つ部分パターン（より短いパターン）が存在するためである．たとえば，4 要素からなるパターン $\langle abcd \rangle$ が存在するとき，3 要素からなる部分パターン $\langle abc \rangle$，$\langle abd \rangle$，$\langle acd \rangle$，$\langle bcd \rangle$ もまた存在する．これらのパターン群を同一グループに含めるため，あるパターン p_1 と p_2 が相互に重なるとき，すなわち，p_1 のインスタンスの少なくとも 1 つの要素が p_2 のインスタンスの要素でもあるとき，それらのパターンは同一のグループとする．

最後に，得られたパターングループの一覧を出力する．このとき，各パターングループ中での，パターンのインスタンス数の最大値を比較基準として，インスタンス数が多いグループから順に出力する．各グループ内のパターンはそれぞれ重複したインスタンスを含んでいるため，並べ替えの基準となるインスタンス数は，合計値ではなく最大値となっている．

4. 適 用 実 験

4.1 実 験 方 法

提案手法の有効性を確認するため，提案手法の一連の手順をツールとして実装し，6 つの Java プログラムに対して $min\_sup = 10$，$min\_len = 4$ という設定で適用した．対象プログラムの名称，バージョン，規模，抽出されたパターン数およびグループ数を**表 2** に示す．

抽出されたパターングループのうち，JDK 標準ライブラリに含まれるメソッド呼び出しのみからなるパターンを除いて，出現回数での上位 5 グループを調査した．出現回数での上位を調査したのは，Marin らの手法において，被呼び出し回数が多いメソッドに横断的関心事との関連性があることが指摘されているためである[21]．抽出されたグループすべてを手作業で調査することは現実的ではなく，調査の労力の都合上，上位 5 件と決定した．

表 2　対象ソフトウェア
Table 2　Target software.

Name	Version	LOC	#Pattern	#Group
JHotDraw	7.0.9	90166	137	37
jEdit	4.3pre10	168335	747	33
Azureus	3.0.2.2	552021	4682	128
Tomcat	6.0.14	313479	1415	85
ANTLR	3.0.1	59687	352	29
SableCC	3.2	35388	162	18

6 つのソフトウェアから上位 5 件のパターングループを選択したところ，Apache Tomcat には出現回数が第 5 位のパターングループが 2 つ存在したため，調査対象となったパターングループの総数は 31（$6 \times 5 + 1$）となった．各グループからは調査対象として，インスタンス数が最大であるパターンと，最長（要素数が最大）のパターンを抽出した．インスタンス数が最大のパターンが，同時にグループ内で最長であるようなグループが 7 個あったため，調査したパターンの数は 55 個（$31 \times 2 - 7$）となった．

4.2 抽出されたコーディングパターン

調査した 55 個のパターンのうち，特定の機能の実装に関わりがあると判断した 33 個を**表 3** に示す．残る 22 個は，アプリケーションの具体的な機能に属すると判断できず，実装におけるイディオム，あるいは無意味なパターンであると判断し，表 3 からは除外した．たとえば，**図 4** は，JHotDraw 7.0.9 から抽出されたパターンで，メソッドの戻り値が `null` かどうかを検査し，`null` でなければもう 1 度そのオブジェクトを取得，処理を続行するという定型的な処理である．

表 3 の *ID* 列は，本文中でパターンを識別する ID である．ID は，ソフトウェア名，パターンが所属するグループの番号，そしてアルファベットからなり，アルファベット “S” は，それぞれグループ中で最もインスタンス数が多い（Supported）パターンを，“L” は最も長い（Longest）パターンを意味する．たとえば，パターン “JHotDraw 8L” は，JHotDraw から検出された，8 番目にインスタンスが多いパターンのグループの中で，最も長いパターンである．*Sup* 列はパターンのインスタンス数（そのパターンを含むメソッドの数）を，*Len* 列はパターンの要素数を，それぞれ示す．*Elements* 列は，パターンの要素を “/” で区切ったリストである．*Type* 列は，複数のプログラムについて調査した結果，複数のプログラムで共通の実装方法だと推測された 4 つのカテゴリの番号を記述している．以下，カテゴリの区分と合わせて，コーディングパターンの例を記述する．

(1) **特定の条件が成立しているとき，複数のメソッド内で追加の処理を行う．**このパターンのカテゴリは，`boolean` を戻り値とするメソッドと，その戻り値によって条件分岐を行う `if` 文，それによって実行される追加の処理が組となっている．具体例としては，パターン Azureus 5S や，Tomcat 1S が該当する．なお，ロギングは横断的関心事の典型例として知られているが，Azureus や Tomcat におけるロギングは多様なメッセージを記録するため，モジュール化は困難である．たとえば，Azureus のソースコードをキーワード検索で

表 3　調査したコーディングパターン
Table 3　Coding patterns in the target software.

ID	Sup	Len	Elements	Type
JHotDraw 8S	19	4	LOOP / willChange / changed / END-LOOP	(3)
JHotDraw 8L	11	5	LOOP / willChange / transform / changed / END-LOOP	(3)
jEdit 1S	55	4	openNodeScope / jjtreeOpenNodeScope / closeNodeScope / jjtreeCloseNodeScope	(3)
jEdit 1L	10	12	openNodeScope / jjtreeOpenNodeScope / jj_consume_token / Expression / jj_consume_token / IF / clearNodeScope / ELSE / popNode / END-IF / closeNodeScope / jjtreeCloseNodeScope	(3)
jEdit 3S	34	4	IF / getToolkit / beep / END-IF	(2)
jEdit 3L	10	6	isEditable / IF / getToolkit / beep / END-IF / remove	(2)
jEdit 9S	25	4	isEditable / IF / beep / END-IF	(2)
jEdit 9L	10	5	isEditable / IF / beep / END-IF / setCaretPosition	(2)
Azureus 2S	151	4	enter / iterator / next / exit	(3)
Azureus 2L	10	10	enter / iterator / hasNext / LOOP / next / IF / add / END-IF / END-LOOP / exit	(3)
Azureus 4S	140	4	size / LOOP / printStackTrace / END-LOOP	(4)
Azureus 4L	10	7	size / LOOP / get / printStackTrace / END-LOOP / size / get	(4)
Azureus 5S	119	4	isEnabled / IF / log / END-IF	(1)
Azureus 5L	10	13	log / isEnabled / IF / log / END-IF / isEnabled / IF / log / END-IF / isEnabled / IF / log / END-IF	(1)
Azureus 6S	97	4	getDataSource / setSortValue / isValid / setText	
Azureus 6L	12	5	getDataSource / setSortValue / isValid / getText / setText	
Azureus 8S	85	4	iterator / hasNext / next / printStackTrace	(4)
Azureus 8L	14	7	iterator / hasNext / next / iterator / hasNext / next / printStackTrace	(4)
Tomcat 1S	304	4	isDebugEnabled / IF / debug / END-IF	(1)
Tomcat 1L	10	24	isDebugEnabled / IF / debug / END-IF / ... (the same sequence is repeated 6 times.)	(1)
Tomcat 6SL	46	4	isPackageProtectionEnabled / IF / doPrivileged / END-IF	(2)
Tomcat 8S	42	4	IF / debug / END-IF / debug	
Tomcat 8L	11	6	IF / debug / END-IF / IF / debug / END-IF	
Tomcat 11S	38	5	isInfoEnabled / IF / getString / info / END-IF	(1)
Tomcat 11L	11	8	getName / getString / isInfoEnabled / IF / getName / getString / info / END-IF	(1)
Tomcat 12S	38	7	createManagedName / findManagedBean / getDomain / IF / getDefaultDomain / END-IF / createObjectName	
Tomcat 12L	19	13	createManagedName / findManagedBean / getDomain / IF / getDefaultDomain / END-IF / createMBean / createObjectName / isRegistered / IF / unregisterMBean / END-IF / registerMBean	
ANTLR 1S	107	4	setErrorListener / newTool / setCodeGenerator / genRecognizer	
ANTLR 1L	10	8	setErrorListener / newTool / setCodeGenerator / genRecognizer / getRecognizer / indexOf / substring / assertEquals	
ANTLR 2S	69	4	setErrorListener / newTool / translate / assertEquals	
ANTLR 2L	10	5	setErrorListener / newTool / translate / assertEquals / checkError	
ANTLR 3S	38	4	LT / match / reportError / recover	(4)
ANTLR 3L	10	11	LA / LT / match / LT / match / LT / match / LT / match / reportError / recover	(4)

```
public class AttributeFieldEventHandler ... {
  protected Set<Figure> getCurrentSelection() {
    if (getCurrentView() != null) {
      return getCurrentView().getSelectedFigures();
    } else {
      return Collections.emptySet();
    }
  }
}
```

図 4　JHotDraw から抽出された null 値チェックのパターン
Fig. 4　A null-check pattern in JHotDraw.

調査したところ，DHTLog.log というメソッドは 55 カ所で呼び出されているが，51 種類の異なるメッセージが引数として使用されており，Logger.log というメソッドの 200 カ所の呼び出しでは 148 種類の異なるメッセージが引数として使用されていた．

(2) **特定の条件が成立しているとき，複数のメソッドの処理を切り替える．**このカテゴリのパターンは，構成要素自体は (1) のパターンと同様であるが，条件分岐の結果がメソッドの処理を完全に切り替えるというものである．たとえば，jEdit 3S，3L，9S，9L のパターン群は，読み取り専用のバッファに対してテキスト編集操作を実行しようとしたとき，beep 音を鳴らして動作をキャンセルする処理を実装している．パターンに該当するソースコードの一部を**図 5** に示す．

(3) **複数のメソッドに共通した前処理と後処理を実行する．**このカテゴリのパターンは，それぞれ異なる処理を実行するメソッド群に含まれる，共通の前処理と後処理であった．たとえば，jEdit 1S では構文解析処理の前後に openNodeScope と closeNodeScope が使用されていた．また，Azureus は，マルチスレッドで処理を実行するため，様々な共有データへのアクセスの前後に AEMonitor.enter と AEMonitor.exit というメソッドを用いた同期処理を挿入しており，それが Azureus 2S，2L というパターンとして抽出された．

(4) **例外に対して，特定の処理を実行する．**このカテゴリのパターンは，一群のメソッドに対して，try/catch ブロックと同時に出現し，共通の例外処理を実装していた．たとえば，パターン Azureus 4S は Debug.printStackTrace というメソッドによって例外発生の状態を記録しており，パターン ANTLR 3S は reportError と recover というメソッドによってエラーへの対応を行っている．

その他の，カテゴリ分類を持たないパターンは，それぞれアプリケーション固有の処理を実装している．たとえば，Tomcat 12L は createMBean という名前を持つメソッド群にのみ存在する，Managed Bean の処理に関連するパターンである．また，Tomcat 8S，8L は，(1) に分類されたロギングの実装と異なり，if 文などをともなわずに debug メソッドを呼び出しているケースであった．

```
public class CompleteWord extends CompletionPopup {
  public static void completeWord(View view) {
    JEditTextArea textArea = view.getTextArea();
    Buffer buffer = view.getBuffer();
    int caretLine = textArea.getCaretLine();
    int caret = textArea.getCaretPosition();
    if(!buffer.isEditable()) {
      textArea.getToolkit().beep();
      return;
    }
    :

public class TextArea extends JComponent {
  public void backspaceWord(boolean eatWhitespace) {
    if(!buffer.isEditable()) {
      getToolkit().beep();
      return;
    }
    :
```

図 5　jEdit から抽出されたパターン 9S のソースコード
Fig. 5　Instances of the pattern 9S in jEdit.

なお，本実験で付与したカテゴリ分類は，発見されたパターン群に基づいて便宜的に定めたものであり，網羅的ではない．また，(1) と (2) は相互に排他的であるが，(3) や (4) とは排他的とは限らない．

4.3　コーディングパターンを用いた保守支援

本実験で調査した 4 つのカテゴリは，いずれもプログラムの複数のメソッドに一貫した振舞いを持たせるためのものである．「一貫した振舞い（Consistent Behavior）」は，Marin らが分類した横断的関心事の 1 つである[22]．一貫した振舞いは，モジュール化されていない機能を実現するための実装上のルールであり，ソフトウェアの変更を行うときには重要となる．たとえば，jEdit に新たな編集機能を追加する開発者にとっては，「バッファが変更不可能であれば beep 音を鳴らして利用者に通知する」といったルールを知っておくことは重要であるし，また，beep 音を鳴らす基準が変化したときは，これらすべてのコードを一貫

した状態に保った変更が必要となる．一方，Tomcat や Azureus においては，新たなコードを追加するごとに，パターンに従って，たとえば `isDebugEnabled` メソッドと `debug` メソッドを組み合わせて新たなログ収集を行うことになる．このように，カテゴリに分類されたコーディングパターンは，実装上の暗黙のルールを表しており，ソフトウェア保守において有用な情報である．これらのパターンの情報は，開発者がソースコードを理解する，あるいは編集する際に開発者が直接読むことができるように，Marin らの提案する横断的関心事のドキュメント記述手法[22] を用いてドキュメント化することが有効であると考えられる．

そのほかにも，コーディングパターンの情報は，以下のような既存の手法への入力データとして使用することが考えられる．

パターンの自動リファクタリング　複数のメソッドの一貫した振舞いは，AspectJ のアドバイスや，デザインパターンの 1 つである Template Method によってリファクタリングできる可能性がある．たとえばカテゴリ (2) には `around` アドバイスを，カテゴリ (1), (3) には `before`/`after` アドバイスを，それぞれ適用することが考えられる．このようなアスペクト抽出の自動的なリファクタリングはすでに研究されており[4),29)]，本研究で得られたコーディングパターンは，これらの手法への入力となりうる．また，カテゴリ (4) についても，例外をアスペクト化するための手順が提案されている[7)]．

パターンに該当するコードの一括編集　すべてのパターンが必ずしもリファクタリングとは限らない．たとえばロギング処理は，「プログラムが何をしているか」を抽象的に表現した文字列をファイルに記録するため，そのような文字列をアスペクト側で自動的に作成することは困難である．リファクタリングが困難なコーディングパターンを開発者がそのまま保守し続ける場合，本手法で得られたパターン情報をもとに，コード片の共通部分をエディタで一括編集する[13)]，あるいはテンプレートを用いたソースコードの同時変更手法[26)] を適用するという対応が考えられる．

現状では，カテゴリが網羅的なものではないため，カテゴリ (1)〜(4) に該当しないパターンであっても，一概に有用でないと判断することはできない．たとえば ANTLR 1S のパターンは，テスト実行用のデータを生成するための手順を表現している．このようなコードは一種の重複コードであり，リファクタリングの対象となりうる[8)] ため，パターンとして情報を提示することは有用であると考えられる．一方で，表 3 からは取り除いた 22 個のパターンは，特定の機能と結び付かなかった．たとえば図 4 のコード片は，単独でも意味の読み取りが容易であり，また，`null` 値の検査が欠落している可能性は，静的解析手法によって検出することが可能である[24)]．このようなパターンは，保守において悪影響を与えるとは考えられないため，パターンマイニングの結果から自動的に除外できることが望ましい．今後，多数のコーディングパターンをカテゴリに自動分類するなどして，カテゴリ (1)〜(4) に該当しないパターンにどのようなものがあるか，調査を行っていく必要がある．

5. 考　　察

5.1 コーディングパターンの自動分類

実験では，抽出したパターンに対し，個々のインスタンスをソースコード上で確認する作業を行った．このとき，パターンの意味を理解するには，以下の情報が有用であった．

- パターンの要素の一覧．
- パターンのインスタンスを持つメソッドの一覧．
- 上記 2 項目の情報の，同グループ内のパターン間での差分．

ソースコードを正規化する段階では，メソッドが属するクラス名を取り除いているが，パターンを理解する段階では，クラス名もまた有用な手がかりであった．また，一部のパターンは，特定の名前を持つメソッドにのみ存在している．たとえば，Tomcat 12L のパターンは，`createMBean` という名前のメソッドにのみ所属していたことから，同メソッド群に固有の実装であることが推測可能であった．

抽出されたコーディングパターンを効果的に分析するための環境を構築するにあたっては，コーディングパターンのインスタンスが配置されているメソッドやクラスに共通の部分文字列が含まれているかどうか，パターンの要素として出現するメソッド呼び出しの名前に何らかの規則が存在するか，といった，命名規則を自動的に調査するツールが有効であろうと考えている．また，コードクローン検出手法に関する研究では，類似したコード片のファイルあるいはディレクトリ単位での分布情報が有用なクローンを抽出するための基準として提案されており[12)]，コーディングパターンに対しても，インスタンスの分布情報に基づくフィルタリングが適用できる可能性がある．

コーディングパターンをカテゴリ分けしていく段階では，ソースコード上の制御フローの参照が必要であった．たとえば機能を挿入するカテゴリ (1) のパターンと機能を切り替えるカテゴリ (2) のパターンは，いずれも `boolean` を戻り値とするメソッドと `if` 文からなっている．制御構造要素をパターンの一部として加えたことで，これらのカテゴリを容易に認識することが可能であった．また，JHotDraw 8S のように 1 つのループ内部で実行される処理と，Azureus 2S のようにループの前後に登場する処理とを区別することができ，後者はカテゴリ (3) へと分類することが可能であった．

パターン情報が与えられたとき，どのカテゴリに属するかを自動的に認識するためには，追加でのメソッド単位での制御フロー解析，データフロー解析が必要である．たとえばカテゴリ (1) と (2) のパターンを認識するには，メソッドの戻り値が `if` 文で使用されるかというデータフロー情報と，`if` 文に関する制御フロー情報が必要となる．制御フロー情報は各メソッドのソースコードから計算できるため，計算コストも少なく，容易に実現可能である．また，メソッド呼び出しの結果が条件分岐に使用されたかどうかを調査するには，局所的な情報に基づく簡易データフロー解析[28] が有効であると考えている．カテゴリ (3) には制御フロー解析で対応でき，また，カテゴリ (4) は `try/catch` ブロックに関する情報を付与するだけで抽出できるため，正規化ルールの拡張のみで対応可能であると考えている．

5.2　ソフトウェア設計のパターンへの影響

本研究は，モジュール化が困難なソースコードを発見することで，それらのコードを同時に変更する必要のある開発者を支援することを目的としている．しかし，ある機能をどのようにメソッドに分割し，それを呼び出すかは，開発者の判断に大きく依存する．提案手法はメソッド単位でのパターンマイニングを行っているため，たとえば，長いメソッドをいくつかの短い作業用メソッドに分割すると，それによってパターンの出現回数に影響が出る可能性がある．ただし，ソースコードが複製されるときはブロックなど意味のあるまとまりが単位となる[17] ことから，複製の単位がコーディングパターンの要素となりやすく，また，そのような複製の単位が，開発者によって異なるメソッドへと分割される可能性は低いと仮定している．なお，設計そのものに問題がある場合については，コーディングパターン分析よりも，たとえば設計の欠陥候補を検出するツール[23] の適用が有効であると考えられる．

5.3　手法の拡張性

本研究では，条件分岐と繰返しのみを制御構造要素としたが，Java における `synchronized` や `try` 構文に対するルールを追加することで，同期処理や例外処理のパターンも発見できる可能性がある．また，識別子の同義語を取り扱える正規化辞書を構築できれば，複数のプログラムを入力としてパターンを抽出し，ソフトウェアのドメイン依存，あるいはソフトウェア開発組織依存のコーディングパターンを抽出できる可能性がある．

本研究で使用した PrefixSpan アルゴリズムは，パターンに該当するコードが 1 つのメソッドに複数回出現する可能性を考慮しない．そのようなコード片を発見したい場合には，コーディングパターンの抽出後，インスタンスを持つメソッドに対して個別に検索を行うことで対応可能である．

PrefixSpan の計算時間は，プログラムに含まれるパターンの総数によって大きく変動する．適用実験のために作成したツールは，Intel Core2 Duo 1.86 GHz の環境で JHotDraw の解析を約 40 秒で完了する一方，Azureus の解析には数時間を要した．より大規模な解析対象に本手法を適用するために，*Parallel Modified PrefixSpan* という，PrefixSpan の拡張アルゴリズムを並列計算する方法[27] のツールへの取り込みを検討している．

6. 関 連 研 究

6.1　アスペクトマイニング

アスペクトマイニング[5),19),21)] は，経験的な指標を用いて，ソースコードから横断的関心事の候補を探索する手法の総称である．

Marin らは，メソッドの被呼び出し回数（Fan-In）の値を用いた解析手法を提案している[21]．この手法の出力結果は，被呼び出し回数の数え方が異なるが，本研究の提案手法において，長さ 1 のパターンのみを抽出した結果とほぼ同じものであり，表 3 に示したパターン中の要素が，それぞれ独立した横断的関心事の候補として出力されると考えることができる．本研究では，複数のメソッド呼び出しの順序情報を提供することで，利用者にとって横断的関心事の選別が容易になっていると考えられる．一方で，本研究の手法は，ロギングのような単一のメソッドのみからなる横断的関心事を見逃す可能性がある．

Breu らは，版管理システムに蓄積された開発履歴を用いて，一貫した振舞いを実装するためのメソッド群を抽出する手法を提案している[5]．このアプローチは，同じ開発者によって追加された，あるいは短期間に連続して追加されたメソッドは，意図的に 1 カ所に配置されたと考える．たとえば Azureus 2S のパターンのように，「enter と exit は一対で使用しなくてはならない」といった実装上の制約を発見するという用途に適している一方で，開発履歴が十分に蓄積されるまでは適用できないという制限もある．

Krinke は，つねにメソッドの先頭あるいは末尾にのみ配置されるようなメソッド呼び出しを抽出し，アスペクトの候補とする手法を提案している[19]．この手法は，AspectJ の `before/after` アドバイスによるアスペクト抽出を前提としており，我々の手法で抽出するパターンのうち，たとえばカテゴリ (2) のような `around` アドバイスでの抽出が可能なパターンはそもそも検出しない．我々の手法が抽出するパターンに，Krinke が用いている位置情報を取り込むことができれば，パターンの自動分類に有効であると考えられる．

6.2　API 使用法のマイニング

提案手法は，パターンマイニング手法をアスペクトマイニングに適用したものであるが，パターンマイニング手法は API の呼び出し順序の解析にも適用されている．Acharya らは，

手続き間の制御フローを解析し，API の呼び出しにおける半順序関係を抽出する手法を提案している[1]．この手法は，半順序関係の候補を削減するために，どの関数群が 1 つのグループであるのかを決定しておく必要があり，異なるグループに属する API 間の関係は発見しない．たとえば，本研究では，Azureus 2S のパターンのように，ライブラリに属する iterator とアプリケーションに属する enter/exit メソッドを組み合わせて使用しているパターンを発見している．Acharya らの手法では，このようなパターンは検出の対象外である．

Xie らは，開発者があらかじめ興味あるライブラリやクラス名をクエリとして入力し，コード検索エンジンから抽出してきたソースコードに対するパターンマイニングを行うツール MAPO を提案している[30]．この手法は，ライブラリを使用するソースコードが多数公開されていることを前提とした方法であり，本研究とは解析対象が大きく異なる．このような解析対象の差がある一方で，本研究は，解析対象の制御構造をパターンの要素として加えた点でも異なっている．制御構造の情報によって，我々のカテゴリ分類 (1)，(2) のようなパターンの識別が可能になり，また，たとえば jEdit 1L のパターンでは，`IF/ELSE/END-IF` 要素によって，「`clearNodeScope` か，`popNode` のどちらか一方のみを呼び出す」という情報も開発者に提示することが可能となっている．制御構造要素の概念は，たとえば `try/catch` 文に対応する要素を追加すると，順序のみからでは判別できない例外処理のパターンにも対応可能であり，手法の拡張性という点でも有意義である．

6.3　コードクローン検出手法

トークン列比較によるコードクローン検出法[15] と比較した場合，本手法で適用したシーケンシャルパターンマイニングは，(1) 多数の複製が存在するものだけが検出対象である，(2) 順序関係を維持される限り，ソースコード上で不連続のコードを発見することができる，(3) パターンのインスタンスを 1 メソッドにつき 1 つしか検出しない，という違いがある．

構文木を比較するコードクローン検出手法[14] やプログラム依存グラフを比較する手法[18] は，文の追加や順序の入れ替えにある程度対応することができるが，コードクローン検出手法のみで横断的関心事の実装を特定することは困難であると指摘されている[6]．これは，いずれのクローン検出手法も，たとえば連続した 30 トークンの一致[15] といったように，一致するコード片のサイズを検出条件としており，短いコード片から構成されるパターンの検出には適していないためである．

図 5 に示すコード例では，`completeWord` と `backspaceWord` という 2 つのメソッドが，それぞれ，バッファが編集可能かどうかを `isEditable` メソッドの呼び出しによって判定し，編集不可能であった場合は `beep` メソッドを呼び出したのち，処理を中断している．このようなコード片のトークン列あるいは構文木を比較したとき，一致するのは `if` 文のみである．また，プログラム依存グラフを比較すると，変数 `buffer` のデータ依存関係が異なる．具体的には，`completeWord` メソッドでは引数である `view` から取得される戻り値であるが，一方の `backspaceWord` ではフィールドとなっている．このような違いによって，これら 2 つのメソッドに埋め込まれたパターンは，コードクローンとして検出されない．なお，適用実験では最小要素数 $min\_len = 4$ としてパターンを抽出しているため，多くのパターンはソースコード上でも短いコード片となっており，上記の例と同様にコードクローン検出手法では検出対象外となると考えられる．このような検出対象の差異から，開発者が複製したソースコードをすべて探索するといった用途に対しては，提案手法とコードクローン検出手法を併用することで相互に弱点を補完できると考えている．

6.4　デザインパターン

本研究では，コーディングパターンを定型的なソースコードと定義した．現状では，特定のソフトウェアに固有のパターンを検出しており，デザインパターン[9] のような一般性は持っていない．しかし，デザインパターンを用いて設計されたソフトウェアの実装は，コーディングパターンの出現に何らかの関係を持っていると考えられる．たとえば，Java 標準ライブラリに含まれた `Iterator` の使用は，`hasNext` メソッド，`next` メソッドや `LOOP` 要素からなるコーディングパターンとして抽出される．また，図 1 のパターンの例は，JHotDraw の様々な編集操作が `Command` パターンとして実装された際に生じたものである．

Gil らは，クラスの実装上の特徴をマイクロパターン（Micro Patterns）と呼び，一部のデザインパターンの実装上の特徴を整理している[10]．また，Hannemann らは，デザインパターンを実装するためのクラスに特定の名前のメソッドが実装されることを利用して，ソースコードからデザインパターンの実装を発見する手法を提案している[11]．

今後，複数のソフトウェアを横断した解析を行うことで，デザインパターンの典型的な実装を行うコーディングパターンを発見できる可能性がある．ただし，デザインパターンの実装に関するメソッドの名称はソフトウェア間で異なっている可能性があるため，たとえば同義語辞書によって対応付けを行うといった提案手法の拡張が必要であると考えられる．

7. ま と め

本研究では，複数のモジュールに分散した定型的なコードを検出するために，Java プログラムを正規化するルールを定義し，シーケンシャルパターンマイニングを適用する手法を提案した．提案手法を 6 つの Java プログラムに対して適用した結果，アプリケーションの

様々な機能を実装するパターンを検出し，パターン情報がソフトウェアを保守する際に有用な手がかりとなることを確認した．

今後の課題としては，開発者が注目したいコーディングパターンだけを絞り込むためのコーディングパターンの自動分類や検索手法の検討，ツールの性能改善があげられる．

謝辞　本研究は，日本学術振興会科学研究費補助金萌芽研究（課題番号：18650006）の助成を得た．また，本研究は Microsoft CORE4 IJARC Project の助成を得た．

参 考 文 献

1) Acharya, M., Xie, T., Pei, J. and Xu, J.: Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications, *Proc. Joint Meeting of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp.25–34 (2007).
2) Agrawal, R. and Srikant, R.: Mining Sequential Patterns, *Proc. 11th International Conference on Data Engineering*, pp.3–14 (1995).
3) Baker, B.S.: A Program for Identifying Duplicated Code, *Computing Science and Statistics*, Vol.6, pp.49–57 (1992).
4) Binkley, D., Ceccato, M., Harman, M., Ricca, F. and Tonella, P.: Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects, *IEEE Trans. Softw. Eng.*, Vol.32, No.9, pp.698–717 (2006).
5) Breu, S. and Zimmermann, T.: Mining Aspects from Version History, *Proc. 21st International Conference on Automated Software Engineering*, pp.221–230 (2006).
6) Bruntink, M., van Deursen, A., van Engelen, R. and Tourwé, T.: On the Use of Clone Detection for Identifying Crosscutting Concern Code, *IEEE Trans. Softw. Eng.*, Vol.31, No.10, pp.804–818 (2005).
7) Filho, F.C., Garcia, A. and Rubira, C.M.F.: Extracting Error Handling to Aspects: A Cookbook, *Proc. 23rd International Conference on Software Maintenance*, pp.134–143 (2007).
8) Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).
9) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
10) Gil, J. and Maman, I.: Micro patterns in Java code, *Proc. 20th Conference on Object Oriented Programming, Systems, Languages and Applications*, pp.97–116 (2005).
11) Hannemann, J., Murphy, G.C. and Kiczales, G.: Role-Based Refactoring of Crosscutting Concerns, *Proc. 4th International Conference on Aspect-Oriented Software Development*, pp.135–146 (2005).
12) 服部剛之，肥後芳樹，楠本真二，井上克郎：コードクローンの分布情報を用いた特徴抽出手法の提案，ソフトウェア信頼性研究会第 3 回ワークショップ論文集，pp.9–17 (2006).
13) Hon, T. and Kiczales, G.: Fluid AOP Join Point Models, *Proc. 2nd Asian Workshop on Aspect-Oriented Software Development*, pp.14–17 (2006).
14) Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: Deckard: Scalable and Accurate Tree-based Detection of Code Clones, *Proc. 29th International Conference on Software Engineering*, pp.96–105 (2007).
15) Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).
16) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming, *Proc. 11th European Conference on Object-Oriented Programming*, pp.220–242 (1997).
17) Kim, M., Bergman, L., Lau, T. and Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL, *Proc. 2004 International Symposium on Empirical Software Engineering*, pp.83–92 (2004).
18) Krinke, J.: Identifying Similar Code with Program Dependence Graphs, *Proc. 8th Working Conference on Reverse Engineering*, pp.301–309 (2001).
19) Krinke, J.: Mining Control Flow Graphs for Crosscutting Concerns, *Proc. 13th Working Conference on Reverse Engineering*, pp.334–342 (2006).
20) Marin, M.: Reasoning about Assessing and Improving the Seed Quality of a Generative Aspect Mining Technique, *Proc. 2nd International Linking Aspect Technology and Evolution Workshop* (2006).
21) Marin, M., van Deursen, A. and Moonen, L.: Identifying Aspects using Fan-in Analysis, *Proc. 11th Working Conference on Reverse Engineering*, pp.132–141 (2004).
22) Marin, M., Moonen, L. and van Deursen, A.: Documenting Typical Crosscutting Concerns, *Proc. 14th Working Conference on Reverse Engineering*, pp.31–40 (2007).
23) Marinescu, R.: Detection Strategies: Metrics-Based Rules for Detecting Design Flaws, *Proc. 20th International Conference on Software Maintenance*, pp.350–359 (2004).
24) Papi, M.M., Ali, M., Correa, Jr., T.L., Perkins, J.H. and Ernst, M.D.: Practical Pluggable Types for Java, *Proc. 2008 International Symposium on Software Testing and Analysis*, pp.201–211 (2008).
25) Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.: PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth, *Proc. 17th*

International Conference on Data Engineering, pp.215–224 (2001).
26) Shonle, M., Griswold, W.G. and Lerner, S.: Beyond Refactoring: A Framework for Modular Maintenance of Crosscutting Design Idioms, *Proc. Joint Meeting of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp.175–184 (2007).
27) Sutou, T., Tamura, K., Mori, Y. and Kitakami, H.: Design and Implementation of Parallel Modified PrefixSpan Method, *Proc. 5th International Symposium on High Performance Computing*, pp.412–422 (2003).
28) Thummalapenta, S. and Xie, T.: PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web, *Proc. 22nd International Conference on Automated Software Engineering*, pp.204–213 (2007).
29) van der Rijst, R., Marin, M. and van Deursen, A.: Sort-based Refactoring of Crosscutting Concerns to Aspects, *Proc. 4th International Linking Aspect Technology and Evolution Workshop* (2008).
30) Xie, T. and Pei, J.: MAPO: Mining API Usages from Open Source Repositories, *Proc. 2006 International Workshop on Mining Software Repositories*, pp.54–57 (2006).

(平成 20 年 7 月 1 日受付)
(平成 20 年 11 月 5 日採録)

石尾　隆（正会員）
平成 15 年大阪大学大学院基礎工学研究科博士前期課程修了．平成 18 年同大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．同年ブリティッシュコロンビア大学ポストドクトラルフェロー．平成 19 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．博士（情報科学）．プログラム解析，アスペクト指向プログラミングに関する研究に従事．日本ソフトウェア科学会，ACM，IEEE 各会員．

伊達　浩典（学生会員）
平成 19 年関西大学総合情報学部総合情報学科卒業．現在，大阪大学大学院情報科学研究科博士前期課程在学中．プログラム解析の研究に従事．

三宅　達也（学生会員）
平成 19 年大阪大学基礎工学部情報科学科卒業．現在，同大学大学院情報科学研究科博士前期課程在学中．アスペクト指向プログラミング，リファクタリングの研究に従事．

井上　克郎（フェロー）
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．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教授．平成 20 年国立情報学研究所客員教授．同年情報処理学会フェロー．同年電子情報通信学会フェロー．工学博士．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．