オープンソースソフトウェアに対するコーディングパターン分析の適用

伊達浩典[†] 石尾隆[†] 井上克郎[†]

コーディングパターンとは，ソースコード中に頻出する定型的なコード片のことである．ロギングや同期処理など，ソフトウェア中でモジュール化することが困難な機能や，プログラミングにおける定型句などが，コーディングパターンとしてソフトウェアから抽出される．

本研究では，開発者が分析したいコーディングパターンのみを自動的に抽出することを目指し，コーディングパターンの特徴の評価尺度として，6つのメトリクスを選定，4つのオープンソースソフトウェアに対して分析を行った．メトリクス間の値の関係と，実際のパターンの特徴を分析した結果，パターンのインスタンス数，インスタンスの分布の広さ，パターンの要素中に含まれる繰り返し構造の比率というメトリクスが，分析すべきパターンの選択にとって有用であることを確認した．

Analyzing Coding Patterns of Open-source Software

Hironori Date,[†] Takashi Ishio[†] and Katsuro Inoue[†]

Coding patterns are idiomatic code fragments. Logging and synchronization are well known features, which are hard to modularize, to be extracted as coding patterns.

In this research, We have selected six software metrics to evaluate coding patterns' characteristics and analyzed coding patterns extracted from four open source programs. This analysis revealed that the number of instances of a pattern, the radius of pattern instances and the ratio of loop elements in a pattern are effective software metrics to select coding patterns worth investigating.

1. まえがき

オブジェクト指向の枠組みでは，モジュール化が困難な機能が存在し，これらの機能の実装はソースコード中に繰り返し登場する[7]．このような機能の代表例としては，ロギングや同期処理が挙げられており，機能に該当するソースコードが複数のモジュールに横断的に出現することから，横断的関心事とも呼ばれる[6]．

このような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，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同様の変更を適用すべきか検討する必要がある[1),2)]．

これまで，我々の研究グループでは，ソースコードに対するパターンマイニングを用いたコーディングパターン検出手法を提案し，いくつかのオープンソースソフトウェアに対して適用を行ってきた[4),9)]．その結果，コーディングパターンには，横断的関心事に該当するパターンだけでなく，ライブラリの定型的な使い方なども含まれていることが判明している．

しかし，大規模なソフトウェアから多数のコーディングパターンが検出されるが，従来はその分析を手作業に頼っていたため，調査可能なパターンの総数が限られていた．Marin らによる，被呼び出し回数が多いメソッドには横断的関心事との関連性があるという指摘[8)]に基づき，パターンに該当するソースコード片の数が多いものほど重要なパターンであると考え，分析対象を選択していた．パターンに該当するソースコード片の数は，しばしば有用なパターンの発見に役立つが，有用でないパターンも多く選択していた．

本研究では，開発者が注目したいパターンのみを効率的に分析可能な環境を構築するため，新たな評価尺度として導入可能なメトリクスの評価を行った．パターンの長さやインスタンス数といった単純なメトリクス以外に，コードクローン検出法[5)]で用いたソースコード片の出現位置に関するメトリクス[3)]についても評価を行った．メトリクス間の値の関係と，実際の

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

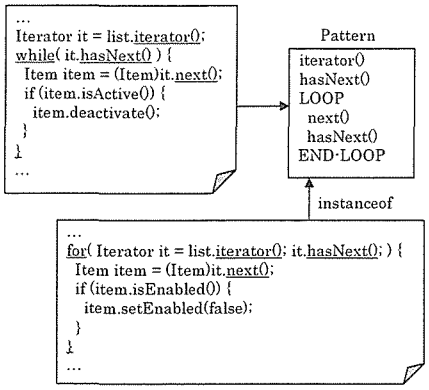

図 1 Iterator を使用したループ処理のパターン

パターンの特徴を分析した結果，パターンのインスタンス数，インスタンスの分布の広さ，パターンの要素中に含まれる繰り返し構造の比率といったメトリクスが，分析すべきパターンの選択に有用であることを確認した．

2. コーディングパターン抽出法

我々は，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の定型的な列と捉えたパターンマイニング手法を提案している[4]．

コーディングパターンの例として，Iterator 使用したループ処理のパターンを図 1 に示す．コレクションから Iterator オブジェクトを取得し，内部の要素をループにより順次処理する，という一連の処理が，Iterator を使用したループ処理として抽出される．

本研究で扱うコーディングパターンは，複数のメソッドに分散したコード片を表現する．コーディングパターンの抽出では，まず解析対象のソースコードをメソッド単位に分割し，各メソッドに対して正規化ルールを適用する．その結果，1 つのメソッドが 1 つの正規化された要素列に対応した，要素列データベースが得られる．この要素列データベースに対してパターンマイニングを適用し，コーディングパターンを抽出する．

本研究では，プログラミング言語 Java で記述されたソースコードを対象としているが，正規化ルールを適宜改変する事で，提案手法を他の言語にも適用できる．

2.1 ソースコードの正規化

ソースコードの正規化では，各メソッドをそれぞれ独立したコード片とみなし，メソッド単位で，メソッド呼び出し要素と制御構造要素の列へと変換する．メソッドの正規化処理では，ソースコードの詳細であるローカル変数や算術演算などを取り除き，「メソッド呼び出し」と「条件分岐」，「繰り返し処理」の 3 つの特徴を正規化して抽出する．条件分岐と繰り返し処理の正規化には，図 2 に示す正規化ルールを使用する．

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

Expression: ( <cond> ) ? <then> : <else>
Sequence:   <cond>, IF, <then>, ELSE, <else>, END-IF

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

Statement:  for ( <decl> : <expr> ) <body>;
Sequence:   <expr>, LOOP, <body>, END-LOOP

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

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

図 2 条件分岐の正規化と繰り返し処理の正規化ルール

メソッド呼び出しの正規化では，ソースコード中のメソッド呼び出し式を，メソッド呼び出し要素へ変換する．メソッド呼び出し要素は，メソッド名，戻り値の型，引数の型名の列を保持する．しかし，メソッドが所属するクラス名は保持しない．メソッド呼び出しは動的束縛により解決されるため，ソースコードから得たクラス名を保持していても，実際に呼び出されるクラス名と一致しないことがある．データフロー解析により動的束縛を解決できるが，計算コストが大きいため，クラス名を持たないという解決策を採用している．この解決策により，継承関係にないクラスに同一のコードが複製されている場合にも対処できる[4]．

条件分岐の正規化では，ソースコード中の if 文と三項演算子を条件分岐として正規化ルールに従い変換する．条件分岐の正規化を行うことで，if 文により表現された条件分岐と，三項演算子により表現された条件分岐を同一視してパターンを抽出できる．

繰り返し処理の正規化では，メソッド中に登場した for 文，while 文，do-while 文を正規化ルールにより変換する．この正規化ルールは，同一のループ構造を異なる制御文により形式で表現した場合も対応できる．

2.2 シーケンシャルパターンマイニング

シーケンシャルパターンマイニングとは，要素列から順序を考慮して頻出部分列を検出する手法である．

本研究では，ソースコードの正規化を行い作成した特徴データベースに対して，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[11] を適用し，コーディングパターンを抽出する．

3. コーディングパターンの分析用メトリクス

過去の研究では，開発者が部品化することが困難な「横断的関心事」に該当する機能を発見するためにコーディングパターンの分析を行ったが，1 つのプログラ

ムからは数百，数千のコーディングパターンが抽出されるため，その分析対象として，インスタンス数が多いパターンから順番に 10 件程度を選択していた[4]．

コーディングパターンを効果的に分析するためには，分析者が注目すべきコーディングパターンやそのインスタンスだけを自動抽出することが重要である．本研究では，その基盤として使用できるソフトウェアメトリクスの候補として 6 種類を選定し，メトリクス間の相関や，コーディングパターンの特徴を調査した．

メトリクス値は，パターン分析に利用できるように，すべて，パターン P を引数に取り，整数あるいは実数値を返す関数 $f(P) : Pattern \rightarrow value$ という形式で定義した．以下，各トリクスの定義を述べるが，パターン P に対して，それらのインスタンス i は $i \in P$ というように集合 P の要素として記載し，パターン P のインスタンス数は $|P|$ という形式で記述する．

3.1 パターン長：LEN（Pattern Length）

コーディングパターン P のパターン長 $LEN(P)$ は，パターン P に含まれる要素数を示す整数値である．

3.2 パターンのインスタンス数：NOI（Number of Instances）

コーディングパターン P のインスタンス数 $NOI(P) = |P|$ は，パターン P に該当する要素列がマイニング対象のソースコード中に出現した回数（メソッド数）を示す整数値である．本研究で使用しているパターンマイニングのアルゴリズム PrefixSpan では，パターン検出のしきい値としても使用される．

3.3 制御構造要素の割合：RCE（Ratio of Control Elements）

コーディングパターン P の制御構造要素の割合 RCE(P) は，パターン P に含まれる全要素数に対する，制御要素数の割合として計算される実数値である．RCE(P) の値が大きいパターンは，メソッド呼び出しを含まない if 文や for 文の単純な入れ子関係を表現している可能性が高いため，経験的なしきい値 RCE(P)$\leq$ 0.7 によりパターンをフィルタリングする．

3.4 パターンの密度：DEN（Density）

コーディングパターン P の密度 DEN(P) は，パターン P の各要素が，ソースコード上でどれだけ密に配置されているかを示す実数値である．具体的には，$DEN(P) = \frac{\sum_{i \in P} DEN_{inst}(i)}{|P|}$ によって計算する．

ただし，$DEN_{inst}(i)$ は，パターンのインスタンスに対して定義される密度である．パターンに該当する要素を含むメソッドの要素列が与えられたとき，$DEN_{inst}(i) = \frac{LEN(P)}{i \text{の末尾要素位置}_{-i} \text{の開始要素位置}_{+1}}$ となる．

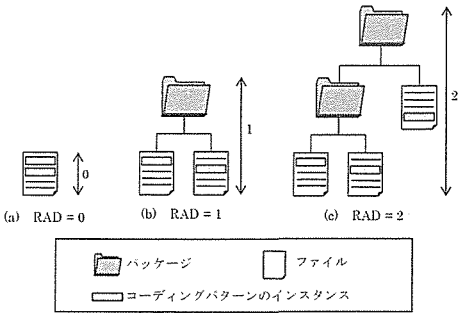

図 3 メトリクス RAD の計算法

3.5 非繰り返し要素の割合：RNR（Ratio of Non-repeated Elements）

パターン P の「非繰り返し」を意味する RNR(P) は，パターン P の要素中の繰り返しを取り除いた後に残る要素の割合を示した実数値である．本研究では，繰り返しの検出には，SEQUITUR アルゴリズム[10] を用いた．このアルゴリズムは，ある要素列が与えられたときに，連続した 2 要素の組が 2 回以上出現した場合を繰り返しとして検出する．8 要素のコーディングパターン「X, A, B, A, B, A, B, Y」中では，「A, B」という組が 3 回繰り返されているため，その 2 回目以降の出現を繰り返しと認識し，RNR は 0.5 となる．

3.6 パターンインスタンスの分散：RAD（Radius）

パターンインスタンスの分散 RAD(P) は，インスタンスが登場するソースコードの範囲を示す整数値である．

RAD は，パッケージ階層中でのパターンインスタンスの分散度合いを表すメトリクスである．図 3(a) では，すべてのインスタンスが同一ファイル内に存在しているため，RAD 値は 0 とする．また，図 3(b) のように，インスタンスが同一パッケージ内の複数のファイルに分散している場合には，RAD 値は 1 とする．さらに，インスタンスが複数パッケージに分散している場合には，それぞれのインスタンスの存在するパッケージをルートノード方向にたどり，すべてのインスタンスを子孫として持つパッケージにたどり着いたら，そのパッケージを基準として，すべてのインスタンスまでの距離を計測し最大のものを RAD とする．図 3(c) の例では，RAD は 2 となる．

4. メトリクスを用いたコーディングパターン分析

調査対象として，図形描画ソフトウェア JHot-

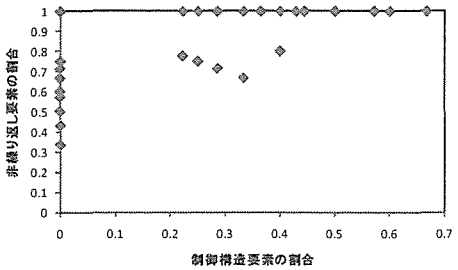

図 4 [JHotDraw] 制御構造要素の割合と非繰り返し要素の割合

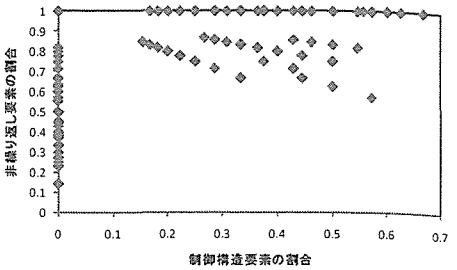

図 5 [jEdit] 制御構造要素の割合と非繰り返し要素の割合

Draw☆1，テキストエディタ jEdit☆2，アプリケーションサーバ Apache Tomcat☆3，パーサジェネレータ SableCC☆4 を用いた．調査対象のソフトウェア一覧と実験に用いたバージョン，ソフトウェアの規模，抽出されたコーディングパターンの数を表 1 に示す．

コーディングパターンを抽出する際のパラメータ設定は，過去の実験の経験により，インスタンス数のしきい値を 10，パターン長のしきい値を 4 とした．

コーディングパターンの特徴を調査するために 3 章で定義した 6 種類のメトリクスを，調査対象から抽出したコーディングパターンに対して適用した．

本研究では，6 種類のメトリクスのすべての組み合わせに対して関連性の調査した．その結果，関連性の認められたメトリクスの組み合わせを本章で述べる．

4.1 非繰り返し要素の割合と制御構造要素の割合

図 4〜図 7 に，X 軸に制御構造要素の割合，Y 軸に非繰り返し要素の割合をとった散布図を示す．

制御構造要素を含まないパターンは，非繰り返し要素の割合に偏りがなく広く分布しているため，制御構造要素を含むパターンのみに着目する．JHotDraw（図 4），jEdit（図 5），SableCC（図 7）に関しては，制御構造要素を含むパターンは，グラフの上部に集中して登場する傾向がある．しかし，Apache Tomcat（図 6）の場合には，制御構造要素を含むパターンの Y 軸方向への偏りは小さい．

表 1 対象ソフトウェア

ソフトウェア	バージョン	LOC	パターン数
JHotDraw	7.0.9	90166	375
jEdit	4.3pre10	168335	2902
Apache Tomcat	6.0.14	313479	8782
SableCC	3.2	35388	450

☆1 JHotDraw, http://www.jhotdraw.org/
☆2 jEdit, http://www.jedit.org/
☆3 Apache Tomcat, http://tomcat.apache.org/
☆4 SableCC, http://sablecc.org/

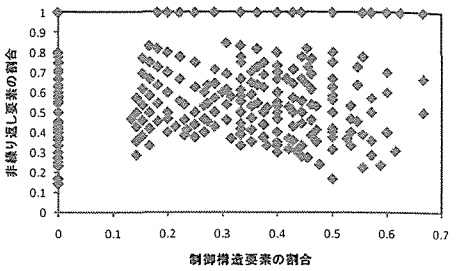

図 6 [Apache Tomcat] 制御構造要素の割合と非繰り返し要素の割合

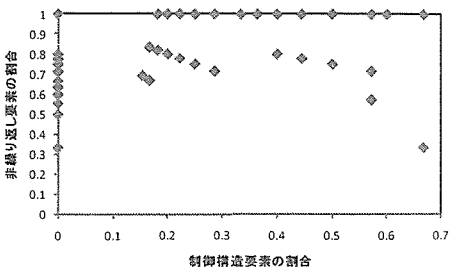

図 7 [SableCC] 制御構造要素の割合と非繰り返し要素の割合

ここで，Apache Tomcat に関する事例についてさらに分析する．制御構造要素は，条件分岐の要素と繰り返し処理の要素の 2 種類に分かれる．そこで，Apache Tomcat から検出されたコーディングパターン中から，条件分岐の要素を含むパターンと，繰り返し処理の要素を含むパターンを別々にプロットした（図 8）．

図 8 の結果では，条件分岐の要素を含む要素は全体に広がっているが，繰り返し要素は，非繰り返し要素の割合が 1 に近い側に偏って分布している．

これらのことから，制御構造要素，特にその中でも LOOP 構造を含むパターンは，非繰り返し要素の割合が高くなる傾向がある．パターン内に LOOP 構造を持つということは，繰り返し処理がその LOOP 構

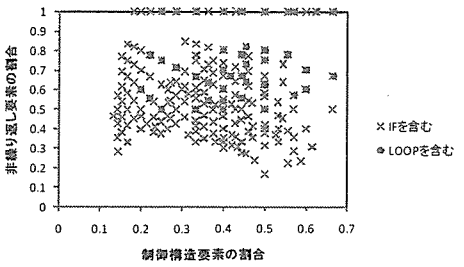

図 8 [Apache Tomcat] 制御構造要素の割合と非繰り返し要素の割合（IF，LOOP 分離版）

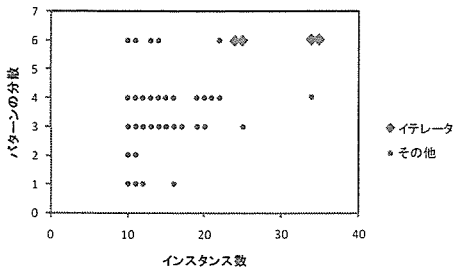

図 9 [JHotDraw] パターンの分散とインスタンス数の関連性

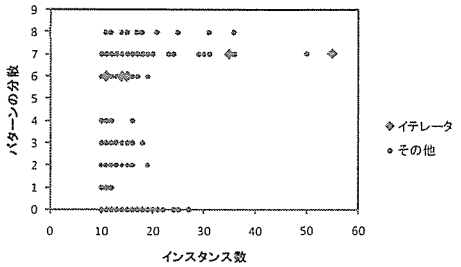

図 10 [jEdit] パターンの分散とインスタンス数の関連性

造により集約されることを意味するので，非繰り返し要素の割合が高くなると考えられる．

図 8 中で，非繰り返し要素の割合が小さく，制御構造要素の割合が大きいパターンは，パターン内に多くの繰り返しを含むため，冗長で保守性が悪い．このようなパターンとして，「isDebugEnabled(), IF, debug(String), END-IF」の要素列が繰り返すパターンがある．このパターンでは，debug メソッドの引数として渡す文字列が，各呼び出しごとに異なり，単純に繰り返し構造を導入するだけでは集約できない．しかし，同一構造の繰り返しが続くためソースコードの可読性が低く改善することが望ましい．

4.2 インスタンス数と出現位置の関連

コーディングパターン分析では，Marin らにより，インスタンス数が多いものほど横断的関心事である可能性が高と指摘されていた[8]．しかし，この指標だけでは，Iterator に代表される，頻繁に使われるライブラリを区別できなかった．本節では，それに代わる指標となりうるパターンの分散 RAD について調査した．

本研究における調査では，パターンのインスタンス数 NOI，パターンの分散 RAD，そしてパターンが Iterator の操作に該当するかどうかを調査した．

Iterator の利用は，Java のプログラマ間では既知の事項であり重要度は低い．しかし，表 2 によると，Iterator を含むパターンの割合は最高で 29.3%に達する．Iterator の利用を取り除く指標を作成することで，プログラム理解に有用な，ソフトウェア固有の機能実装のようなパターンを発見しやすくできる．

Iterator の利用では，メソッド名に「next」や「hasNext」という特徴的な文字列を持つメソッド呼び出している．また，Iterator の利用では，複数のオブジェクトに対して処理を繰り返す必要がある．そこで，コーディングパターン中から，次の条件を満たすものを Iterator を利用しているパターンとして抽出した．

- メソッド名に「next」を含むメソッド呼び出しと，「hasNext」を含むメソッド呼び出しを持つ．
- 制御構造として，繰り返し処理を含む．

その結果，抽出されたパターン数とコーディングパターンの総数に対する割合を表 2 に示す．

また，コーディングパターンを Iterator の利用パターンと，Iterator の利用に関係がないその他のパターンに分類し，図 9〜図 12 にパターンの分散とインスタンス数の関連を示した．

図 9，10 に示すように，JHotDraw と jEdit に含まれる，Iterator を利用しているパターンは，パターンの分散が大きい．また，Iterator を利用したパターンの中にもインスタンス数が多いものが存在する．

図 12 に示した SableCC の場合は，パッケージ階層が浅く，1 つのパッケージに多数のソースファイルが格納されていた．そのため，パターンの分散については，パターンの種類間での差異が判断できなかった．

図 11 に示した Apache Tomcat では，傾向はみられなかった．これは，Iterator とプログラム固有の機

表 2 Iterator の利用を含むパターンの数

ソフトウェア	パターン数	Iterator 利用	割合
JHotDraw	375	8	2.1%
jEdit	2902	28	0.9%
Apache Tomcat	8782	434	4.9%
SableCC	450	132	29.3%

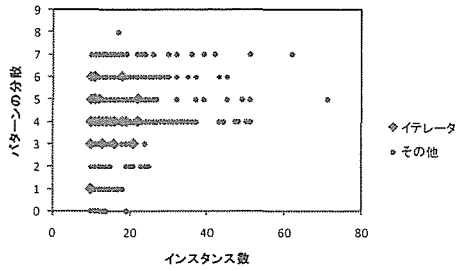

図 11 [Apache Tomcat] パターンの分散とインスタンス数の関連性

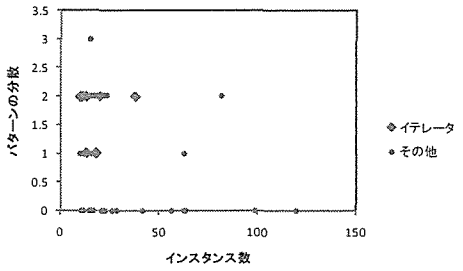

図 12 [SableCC] パターンの分散とインスタンス数の関連性

能実装の両方を含むパターンが，パッケージ階層中で局所的に現れていることが原因と考えられる．

コーディングパターンの従来の調査手法では，インスタンス数が多いものから調査を行っていた．しかし，JHotDraw や jEdit では，Iterator を利用しているパターンがインスタンス数の上位に登場し，有益なパターンが多数のパターン中に埋もれてしまう．この問題を回避するには，パターンの分散が小さいパターン，つまり，特定のパッケージやファイルにのみ登場するパターンから調査するという方法をとる必要がある．

5. ま と め

コーディングパターンの検出結果は膨大になり，有用なパターンを発見することが困難となっている．これを解決するために，コーディングパターンを分類し，閲覧者が必要なものを選び出し提示する必要がある．

そこで，本研究では，コーディングパターンの特徴を計測するためのメトリクスを提案し，その特徴間の関連と，コーディングパターンの種類との関係について分析を行った．その結果，パターンのインスタンス数，インスタンスの分布の広さ，パターンの要素中に含まれる繰り返し構造の割合といったメトリクスが，分析すべきパターンの選択に有用であると確認できた．

謝辞 本研究は，日本学術振興会科学研究費補助金基盤研究（A）（課題番号:21240002）および文部科学省科学研究費補助金若手研究（B）（課題番号:21700030）の助成を得た．

参 考 文 献

1) Baker, B.S.: A Program for Identifying Duplicated Code, *Computing Science and Statistics*, Vol. 6, pp. 49–57 (1992).
2) Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).
3) 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).
4) 石尾隆, 伊達浩典, 三宅達也, 井上克郎: 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, 情報処理学会論文誌, Vol. 50, No. 2, pp. 860–871 (2009).
5) Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670 (2002).
6) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming, *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp. 220–242 (1997).
7) Marin, M.: Reasoning about Assessing and Improving the Seed Quality of a Generative Aspect Mining Technique, *Proceedings of the 2nd International Linking Aspect Technology and Evolution Workshop* (2006).
8) Marin, M., van Deursen, A. and Moonen, L.: Identifying Aspects using Fan-in Analysis, *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 132–141 (2004).
9) Miyake, T., Ishio, T., Taniguchi, K. and Inoue, K.: Towards Maintenance Support for Idiom-based Code Using Sequential Pattern Mining, *Asian Workshop on Aspect-Oriented Software Development(AOASIA3)* (2007).
10) Nevill-Manning, C. and Witten, I.: Identifying Hierarchical Structure in Sequences: A linear-time algorith, *Journal of Artificial Intelligence Research*, pp. 67–82 (1997).
11) Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.: PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth, *Proceedings of the 17th International Conference on Data Engineering*, pp. 215–224 (2001).