修士学位論文

題目

オブジェクト指向プログラムに含まれる コーディングパターンの特徴と出現位置の関連性分析

指導教員
井上 克郎 教授

報告者
伊達 浩典

平成 21 年 2 月 9 日

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 ソフトウェア工学講座

平成 21 年度 修士学位論文

オブジェクト指向プログラムに含まれるコーディングパターンの特徴と出現位置の関連性分析

伊達 浩典

内容梗概

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．

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，分析を行ってきた．しかし，抽出されたコーディングパターンの数に対して，手作業で調査可能なコーディングパターンの総数は限られているため，大規模ソフトウェアなどに対する十分な分析は行うことができなかった．

そこで，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ったメトリクスなどが，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．

主な用語

コーディングパターン
ソフトウェアメトリクス
プログラム理解

目 次

1　まえがき

近年，ソフトウェア開発へのオブジェクト指向プログラミングの採用が増加している．オブジェクト指向の特徴である継承や多態性の仕組みを利用することで，ソフトウェアの再利用性や保守性を向上させることができる．しかし，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 [13]．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 [11].

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 [2, 3].

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 [20, 15, 7]．その結果，コーディングパターンには，プログラムの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は多数のコーディングパターンが検出される一方で，従来はその分析を手作業に頼っていたことから，調査可能なパターンの総数がきわめて限られていた．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 [14] に基づき，パターンに該当するソースコード片の数（インスタンス数）が大きいものほど重要なパターンである，という一元的な評価尺度により，分析対象のパターンを選択していた．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，言語仕様あるいはコーディングスタイルなどから，偶然パターンに該当するコード片が発生することもあり，意味のないパターンを手動で取り除く必要もあった．

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，1 パターンに含まれる要素の種類の数など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 [10, 12] 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 [6] 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などが，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．

以降，2 章では，コーディングパターンとコーディングパターンの検出手法について述べる．次に，3 章では，コーディングパターンの特徴，出現位置について述べる．そして，4 章では，行った実験とその結果についての述べる．最後に，5 章で，まとめと今後の課題を示す．

2　コーディングパターン

コーディングパターンとは，複数のモジュールに分散した定型的なコードである．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（条件分岐と繰り返し）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 [7].

2.1　コーディングパターンの例

コーディングパターンは，その出現するメソッド呼び出し要素の種類によって，大きく 2 種類に分けられる．1 つは，解析対象ソフトウェアの外部で定義されているメソッド，すなわちライブラリを使用することを主とするパターンである．もう 1 つは，解析対象ソフトウェア中の特定の機能を実現するために，ソフトウェア内で定義されているメソッドを呼び出しているものである．

2.1.1　イテレータを用いたループ処理のパターン

Java では，コレクションオブジェクトに含まれる各要素に対して処理を行うために，デザインパターン [4] の一種である Iterator パターンを利用できる．

Iterator パターンの利用は，次の手順で行われる．

1. コレクションオブジェクトから，繰り返し処理のための iterator オブジェクトを取得する．
2. 処理を行う要素がコレクション内に残っているかを調査する．
3. コレクションから要素を取り出し処理を行う．

これらの一連の処理が，図 1 に示すように Iterator を使用したループ処理のパターンとして抽出される．

繰り返し処理は，実際のソースコード上では，for 文や while 文として記述されるが，繰り返し処理の正規化処理により,「LOOP」と「END-LOOP」の組により表現される．

2.1.2　Undo 機能の実現に関するパターン

図 2 は，図形エディタ JHotDraw 5.4b1 から抽出された Undo パターンである．

この Undo パターンは，JHotDraw 5.4b1 のユーザが行った操作を元に戻す「Undo 処理」を実現するパターンである．Undo パターンのインスタンス部分で実際に行っ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

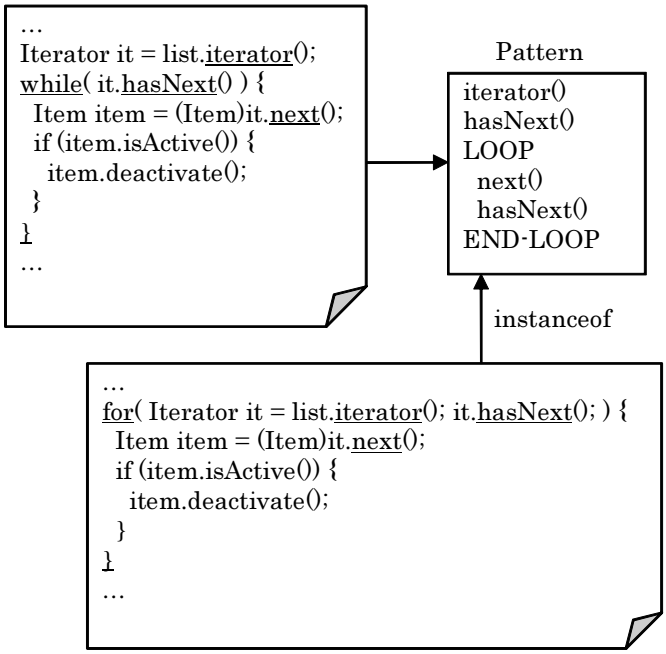

図 1: Iterator を使用したループ処理のパターン

しパターンを開発者が知ることは，この「Undo 処理」がどのように実装されているかを理解するために有用である．また，新たな編集操作を実装するときに役に立つ．

2.2 コーディングパターン抽出法

本節では，コーディングパターン抽出法について述べる．

コーディングパターン抽出の流れを図 3 に示す．

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，特徴シーケンスを抽出する単位であるメソッドに分割する．次に，分割されたそれぞれのメソッド中から特徴情報を抽出し，特徴データベースへ登録する．抽出する特徴情報は，以下のとおりである．

- メソッド呼び出し情報
- 条件分岐情報
- 繰り返し情報

そして，特徴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．

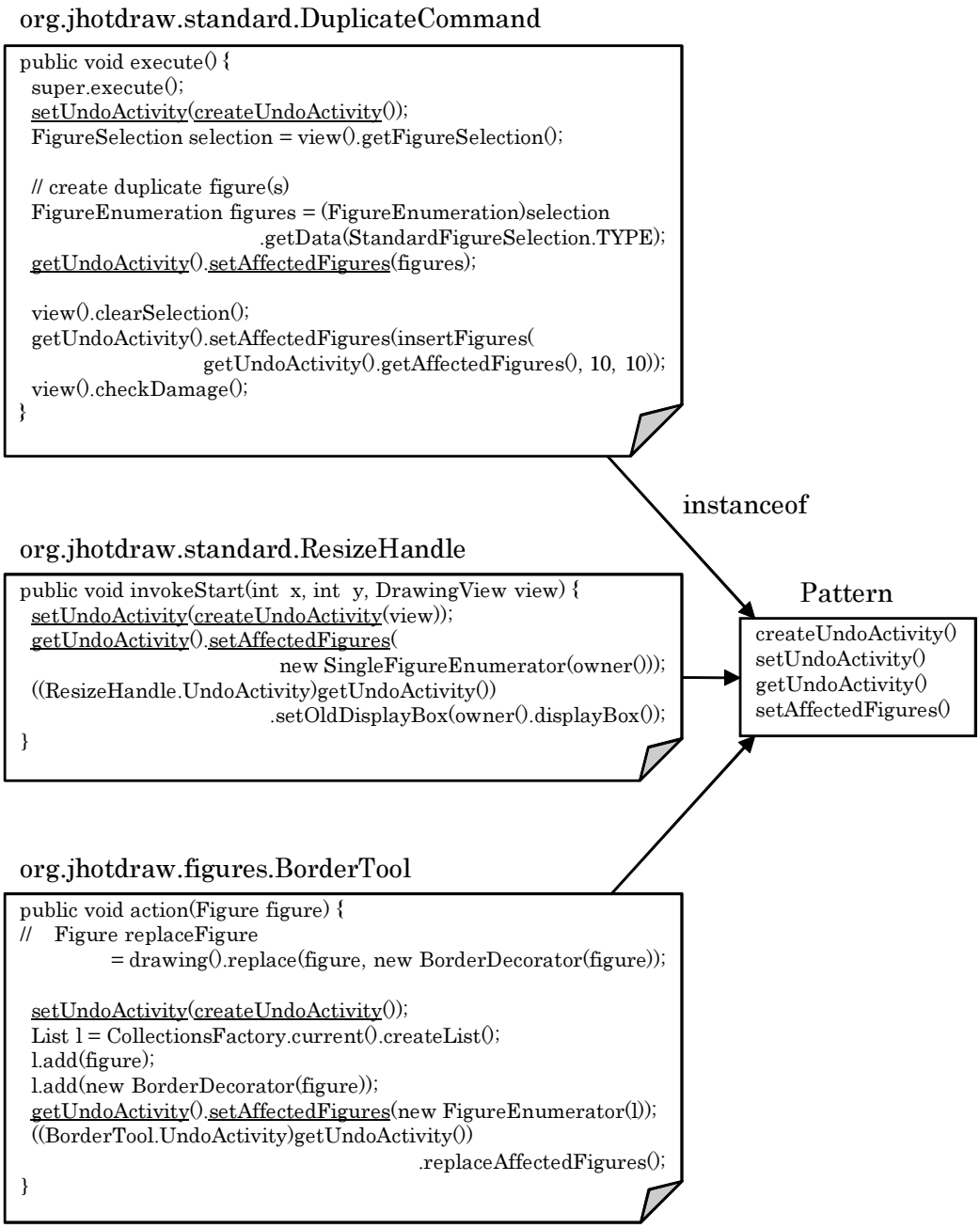

図 2: JHotDraw 5.4b1 から抽出された Undo パターン

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から，コーディングパターンの検出を行う．

他の言語で記述されたソースコードからコーディングパターン抽出を行う場合には，その言語に対応したソースコードの正規化規則を作成する必要がある．

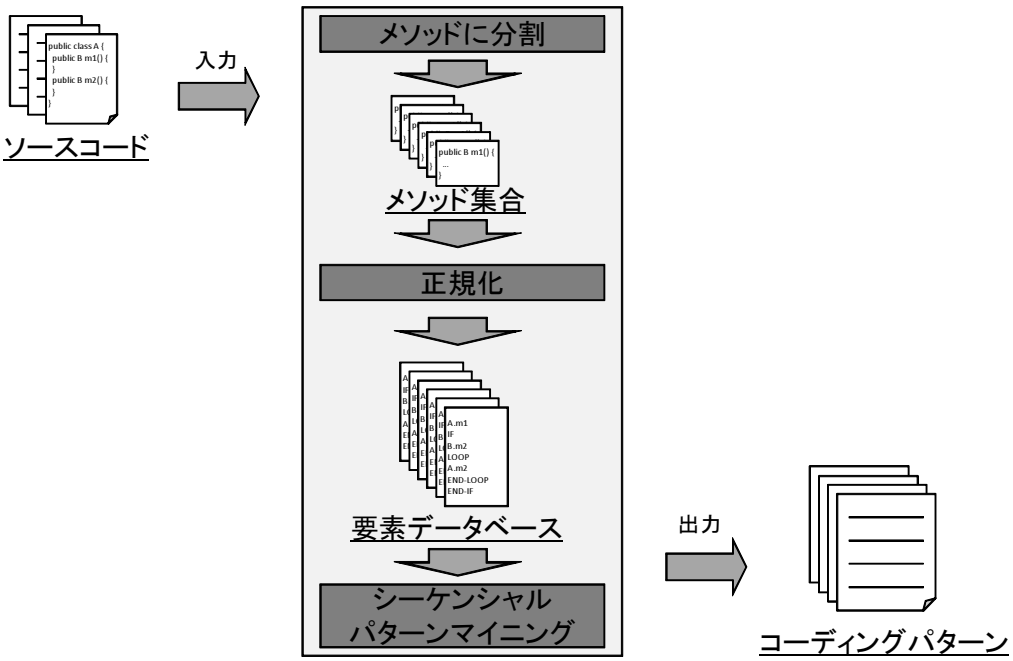

図 3: コーディングパターン抽出の流れ

2.2.1 ソースコードの正規化

ソースコードの正規化では，Java プログラムの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．1 つの Java プログラムは，一般に多数のメソッドから構成されているため，この正規化によって，Java プログラムは，要素列のデータベース（Sequence Database）へと変換される．この要素列データベースが，次のステップであるパターンマイニングの対象となる．

メソッドの正規化処理は，次の 3 つの正規化によって構成される．

- メソッド呼び出しの正規化
- 条件分岐の正規化
- 繰り返し処理の正規化

これらの正規化処理について順に説明する．

メソッド呼び出しの正規化

ソースコード中に登場するメソッド呼び出し式を，メソッド呼び出し要素へと変換する．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．

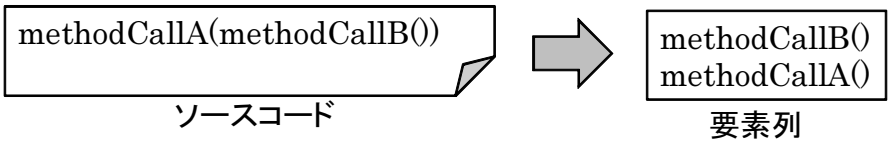

図 4: 引数中でのメソッド呼び出し

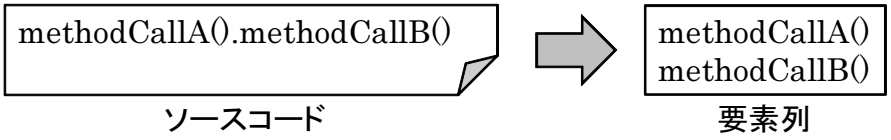

図 5: 戻り値に対するメソッド呼び出し

ここで重要な点は，メソッドが所属するクラス名を持たないことである．Java プログラムでは，メソッド呼び出しは動的束縛によって解決される．すなわち，あるクラス A のメソッド m を呼び出す，とソースコード上に記述されているとき，A を継承したサブクラスのメソッド m が実行される可能性がある．そのため，ソースコードから得られるクラス名を保持していても，実際に呼び出されるクラスの名前とは一致しないことがある．データフロー解析を用いると，実際に起こりうる動的束縛を解決することができる [19] が，このような計算は，プログラム全体のソースコードを解析する必要があるため，計算コストが大きい．そこで，クラス名を持たないという解決策を採用している．なお，クラス名を持たないことは，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という点で有益である [7].

メソッド呼び出しは，構文上は，式（Expression）であり，他の式の部分式として登場しうる．これに対しては，以下の 2 つのルールを用いて対処する．

- メソッド呼び出し要素の順序は，原則として，演算子の優先順位などによって定まる式の評価順序に従う．
- 式の評価順序が言語仕様で定義されていないとき，ソースコードの配置順序に従う．

たとえば，図 4 のように,「methodCallA()」の引数が「methodCallB()」であるときは,「methodCallB を呼び出して，その戻り値を methodCallA に渡す」という順序が規定されている．そのため，正規化された要素列は「methodCallB, methodCallA」となる．

図 5 のように,「methodCallA()」の戻り値として得られたオブジェクトに「methodCallB()」を呼び出す場合は，正規化された要素列は「methodCallA, methodCallB」となる．

一方，図 6 のように，methodCallA と methodCallB の結果を単純に加算している場合，どちらを先に呼び出すかは規定されていない [5]．本研究では，このときは，ソースコード

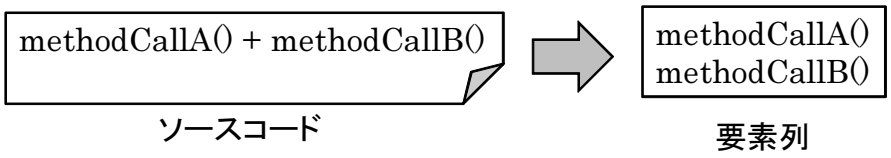

図 6: 式中の複数メソッド呼び出し

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

図 7: 条件分岐の正規化ルール

上で先に登場する methodCallA のほうが先に呼び出されると考え，メソッド呼び出し要素を配置する．

条件分岐の正規化

ソースコード中の，if 文と三項演算子は条件分岐として正規化する．条件分岐の正規化規則を図 7 に示す．条件分岐の正規化を行うことで，パターンマイニングの段階で if 文によって表現された条件分岐と，三項演算子によって表現された条件分岐を同一視してパターンの抽出を行うことができる．

Java の論理演算子，`&&` や `||` は，短絡評価を引き起こす．たとえば 2 つの式 a, b を `a && b` というように連結した場合，式 a が偽であったときは，式 b を評価しない．これは条件分岐の一種とも考えることができるが，`if` 文などの条件式に頻繁に出現する記法であることから，条件分岐ではない単純な式と同様に扱い，正規化の結果には影響を与えない．

繰り返し処理の正規化

メソッド中に登場した for 文，while 文，do-while 文を正規化する．繰り返し処理の正規化規則を図 8 に示す．

制御構造要素 LOOP，END-LOOP は，繰り返し実行される処理の範囲に対応する．たとえば for 文の場合，繰り返し実行の対象である body 部を実行し，カウンタのインクリメントなどが記述される update 部を実行したのち，条件式が評価される，という一連の処理がループ 1 回の実行単位であると考えている．この正規化ルールは，まったく同一のループ構造を，for や while など，異なる制御文を使って書き直した場合に対応するよう定義している．

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 8: 繰り返し処理の正規化ルール

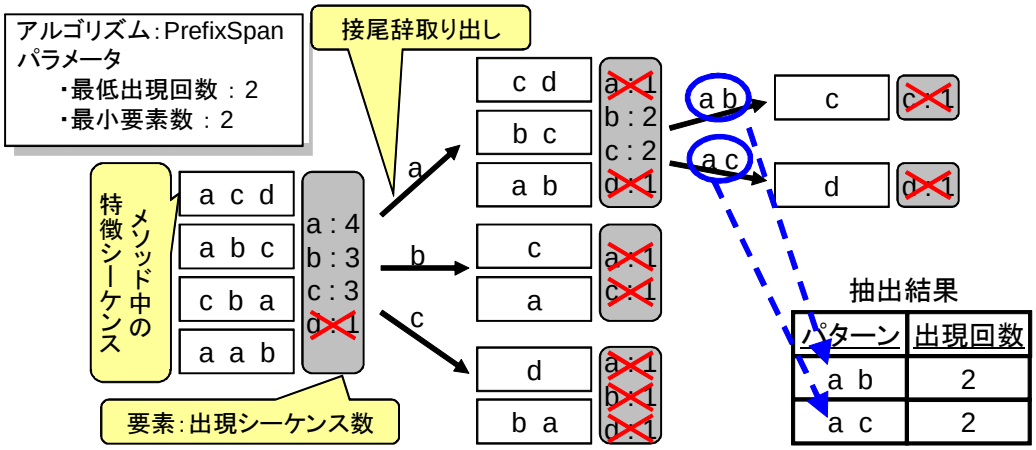

図 9: シーケンシャルパターンマイニング

その他の構文

Java には，メソッド呼び出し，条件分岐や繰り返し以外にも，たとえば switch 文のような特殊な条件分岐や try-finally 文のような制御フローに影響を与える構文が存在する．しかし，これらの構文には，現状では対応していない．これらの構文が出現した場合は，その中に含まれている文がそのまま並べられているものとして，構文自体を無視している．

2.2.2 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，マイニング対象の要素列から，要素の順序を考慮したパターン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[17] を適用し，コーディングパターンを抽出する．

シーケンシャルパターンマイニングの処理の例を図 9 に示す．この図では，4 つの特徴シーケンス 「a, c, d」,「a, b, c」,「c, b, a」,「a, a, b」から，最低 2 回出現する，長さ（要素数）2 以上のパターンを検出している．

シーケンシャルパターンマイニングでは，まず，シーケンス中に登場するすべての要素に対して，要素が出現するシーケンス数を調査する．図 9 の例では，要素 a は 4 つのシーケンスに登場している．また，要素 b, c, d はそれぞれ，3, 3, 1 個のシーケンスに出現する．ここで，最低 2 回出現するという条件に従って，フィルタリングを行う．ここでは，a, b, c それぞれが条件を満たしており，長さ 1 のパターンとして認識される．

続いて，長さ 1 のパターンに対し，接尾辞の取り出し（projection）を行う．要素「a」が頻出しているということから，各シーケンスの最初の a の出現までを取り除いた，接尾辞データベース（projected database）を構築する．この接尾辞データベース上で頻出する要素を数えると，b, c が 2 回ずつ出現していることがわかる．このことから,「a, b」「a, c」はいずれも 2 回出現した長さ 2 のパターンとして認識することができる．

この接尾辞の取り出し操作を繰り返すことで，長さ k のパターンから長さ k+1 のパターンを構築していき，最終的に新たなパターン候補が発見されなくなった時点で計算を終了する．

このようにして得られるパターンは，次のような性質をもつ．

- パターンは，メソッド呼び出し要素と制御構造要素の順序をもった列である．
- パターンは，シーケンス上での出現（インスタンス）においては，連続した要素であるとは限らない．
- 長さ k+1 のパターンが検出されているとき，そこから 1 要素だけを取り除いた長さ k のパターンもまた検出されている．たとえば「a, b, c, d」というパターンが検出されている場合,「a, b, c」や「a, b, d」といったパターンも検出される．

2.2.3 不要パターンの除去

パターンマイニングの結果には，パターンとして不適切なものが含まれている．以下のようなパターンは，不適切であると判断し，パターンマイニングの結果から取り除く．

制御構造の開始要素と終了要素の間にメソッド呼び出しが含まれないパターン

制御構造は，その制御構造が関連しているメソッド呼び出しを実行するかどうかを決定したり，関連しているメソッド呼び出しを繰り返し実行するというように，メソッド呼び出しに関連して，初めて意味を持つので，制御構造の開始要素と終了要素の間にメソッド呼び出しが含まれないパターンは，不適切であるため結果から取り除く．

制御構造の割合が 70%を超えるパターン

パターンの構成要素中の制御構造の割合が，メソッド呼び出しに関連したパターンである可能性は低いため削除する．

この 70%という割合は，経験的に定めたものである．

制御構造の開始要素と終了要素の対応がとれていないパターン

制御構造の開始要素と終了要素は，必ず対応して登場するため，この対応関係の崩れたパターンは意味をなさないため削除する．

ELSE 要素が，IF 要素と END-IF 要素に囲まれていないパターン

ELSE 要素は，その ELSE 要素が対応する，IF 要素と END-IF 要素の間に登場しなければ意味を成さない．そこで，単体で登場する ELSE 要素を含むパターンは，削除する．

3　コーディングパターンの特徴と出現位置

コーディングパターンは，ソースコードに頻出するメソッド呼び出しの列である．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，その分析対象はインスタンス数が多いパターンから順番に選んだ，ごく少数のパターンに限られていた [7]．インスタンス数が多いパターンを優先的に調査したのは，パターン中で呼び出されているメソッドは被呼び出し回数が多いという事実と，横断的関心事は被呼び出し回数が多いメソッドによって構成されていることが多いという Marin らの指摘に従ったものである [14]．

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的に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することができるソフトウェアメトリクスの候補を選定し，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．

コーディングパターンから得られる情報には，大きく分けて以下の 2 種類がある．

- パターン自身の情報．パターンに含まれる要素数や，パターンのインスタンス数がこれに該当する．
- パターンのインスタンスから得られる情報．パターンに該当するソースコード片の位置などはこちらに含まれる．

本研究では，これらの情報を表現したメトリクス値として 6 種類を選定した．パターンの分析を行うという目的から，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，それらのメ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ものとする．

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 に含まれている要素の数を示す整数値である．

コーディングパターン検出ツールにより，パターン長のしきい値に満たないパターンは，検出結果から取り除かれている．

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 を構成する要素列が，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，しきい値としても用いられている．

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 P 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だけを表現している可能性が高い．そのため，パターンマイニングの終了後，経験的に定めたしきい値 RCE(P) ≤ 0.7 という条件で，パターンのフィルタリングを行っている．

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，以下の式によって計算する．

$$DEN(P) = \frac{\sum_{i \in P} DEN_{inst}(i)}{|P|}$$

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，

$$DEN_{inst}(i) = \frac{LEN(P)}{i \text{ の末尾要素の位置} - i \text{ の開始要素の位置} + 1}$$

インスタンスの密度の計算例を図 10 に示す．図 10 右に示されているコーディングパターンが，図 10 左のようなインスタンスとして出現しているとき，先頭要素 iterator() から末尾要素 END-LOOP まで，パターンに該当する 6 要素が 12 要素の列中に出現しているとみなし，このインスタンスに対する密度は $DEN_{inst}(i) = 6/(12 - 1 + 1) = 0.5$ となる．

パターン P の密度 DEN(P) は，P のすべてのインスタンスの密度の平均として定義されており，各インスタンスが他の要素を間に含まない単一のコード断片に近づいていくほど，値が 1 に近づいていく．

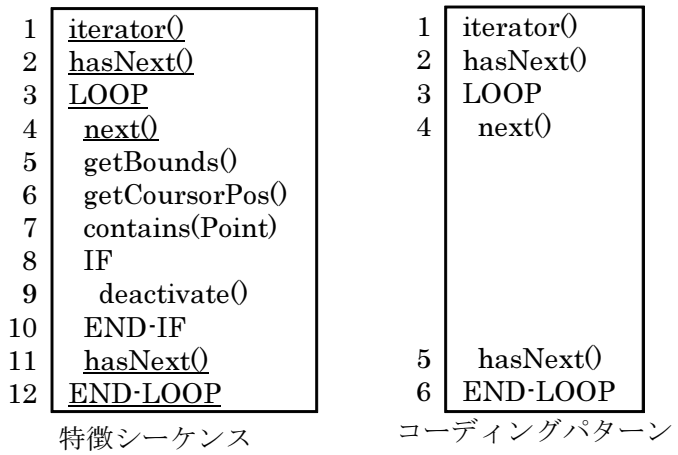

図 10: メトリクス DEN

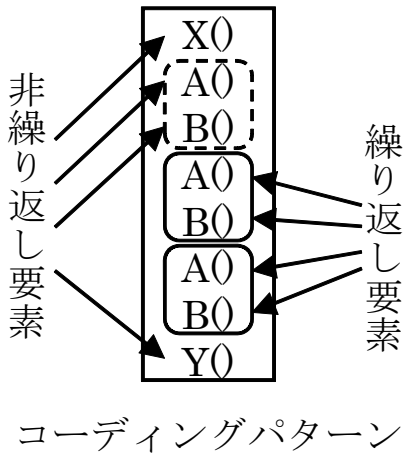

図 11: メトリクス RNR

3.5 非繰り返しの要素割合：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 の要素中に含まれる繰り返し構造を検出，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 [16] 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．図 11 は，8 要素のコーディングパターン「X, A, B, A, B, A, B, Y」が与えられたときの非繰り返し要素を示している．このパターンは,「A，B」という組が 3 回繰り返されていることから，その 2 回目以降の出現を繰り返し要素と認識し，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，パターン P のインスタンスが配置されているソースコードの範囲を示す整数値である．

Java 言語では，ソースコードをパッケージという階層構造によって管理しており，開発組織のドメイン名とソフトウェア名，さらにソフトウェア中のサブシステム名などを用いて

```
jp.ac.osaka_u.ist.sel.pattern_mining
jp.ac.osaka_u.ist.sel.pattern_mining.metrics
org.eclipse.jdt.core
```

のように，重複しない名称を使用する [5].

RAD は，パッケージ階層中でのコーディングパターンのインスタンスの分散度合いを表すメトリクスである．図 12(a) では，コーディングパターンのすべてのインスタンスが，同一のファイル内に存在しているため，RAD 値は 0 とする．また，図 12(b) のように，コーディングパターンのインスタンスが，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，コーディングパターンの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に向かってたどり，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 12(c) の例では，RAD は 2 となる．

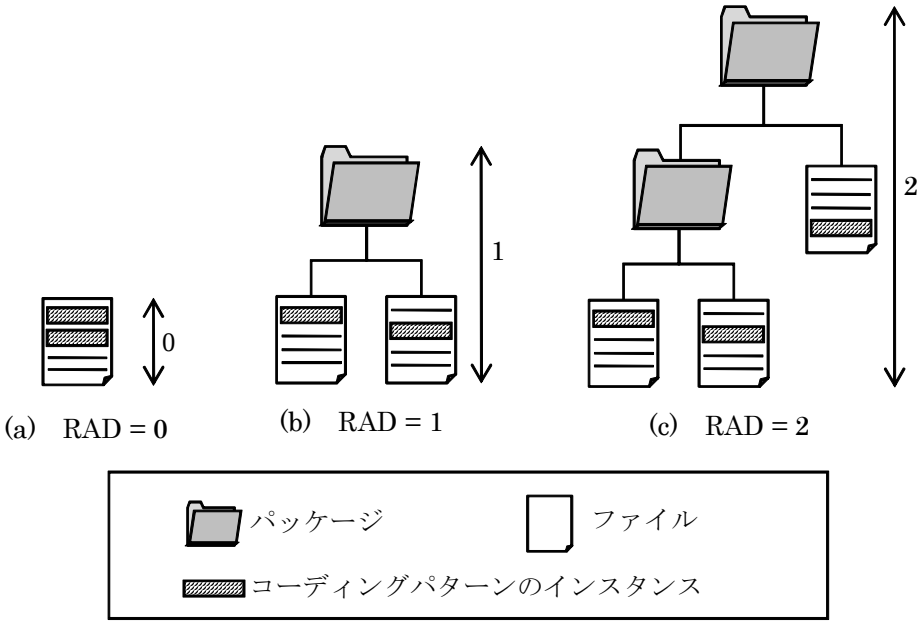

図 12: メトリクス RAD

4　メトリクスを用いたコーディングパターン分析

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，オープンソースソフトウェアから抽出したコーディングパターンに対して適用した．

4.1　対象ソフトウェア

調査対象として，図形描画ソフトウェア JHotDraw[9]，テキストエディタ jEdit[8]，アプリケーションサーバ Apache Tomcat[1]，パーサジェネレータ SableCC[18] を用いた．

調査対象としたソフトウェアの一覧と実験に使用したバージョン，ソフトウェアの規模，実際に検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，次の通りである．

- インスタンス数のしきい値：10
- パターン長のしきい値：4

4.2　メトリクス間の関連分析

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を行った．本節では，関連性の認められたメトリクスの組み合わせについて述べる．

4.2.1　非繰り返し要素の割合と制御構造要素の割合

制御構造要素を含むパターンと制御構造要素を含まないパターンそれぞれの数を表 2 に示す．

制御構造要素を含むパターンと含まないパターンの数を比較した場合，今回解析したソフトウェアすべてで，制御構造要素を含むパターンの数が制御構造を含まないパターンの数より多く検出されている．これは，制御構造要素の種類が IF，ELSE，END-IF，LOOP，

表 1: 対象ソフトウェア

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

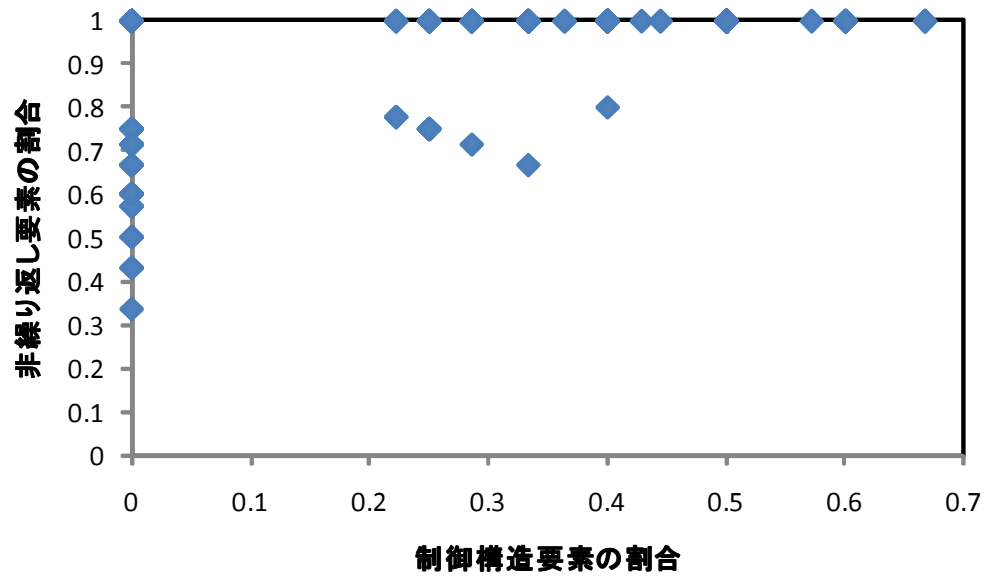

図 13: [JHotDraw] 制御構造要素の割合と非繰り返し要素の割合

END-LOOP の 5 種類と少なく，プログラムを作成するために多用されるため，パターンとして検出されやすいことが要因として考えられる．

図 13〜図 16 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パターンが検出されたソフトウェアごとに示す．

図 13〜図 16 中で，制御構造要素を含まないパターンは，Y 軸上に配置される．また，制御構造要素を含んでいるパターンは，制御構造要素の割合に応じてグラフ上に表示されている．

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 13），jEdit（図 14），SableCC（図 16）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

表 2: 制御構造を含むパターン含まないパターンの数

ソフトウェア名	制御構造要素無し	制御構造要素有り
JHotDraw	140	235
jEdit	1212	1690
Apache Tomcat	2489	6293
SableCC	120	330

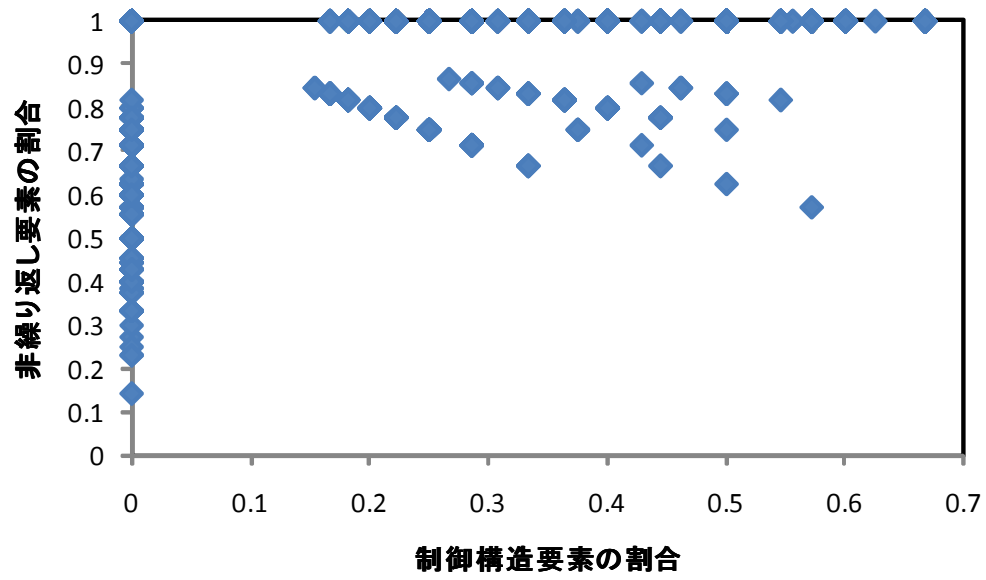

図 14: [jEdit] 制御構造要素の割合と非繰り返し要素の割合

場する傾向がある．しかし，Apache Tomcat（図 15）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．

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（IF，ELSE，END-IF），繰り返し処理の要素（LOOP，END-LOOP）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んでいるパターンと，繰り返し処理の要素を含んでいるパターンを別々にプロットした．条件分岐の要素，繰り返し処理の要素を別々にプロットした散布図を図 17 に示す．

図 17 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．

これらのことを総合して考えると，制御構造要素，特にその中でも LOOP 構造の要素を含むパターンは，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 構造により集約さることを意味するので，繰り返される要素が減少し，非繰り返し要素の割合が高くなると考えられる．

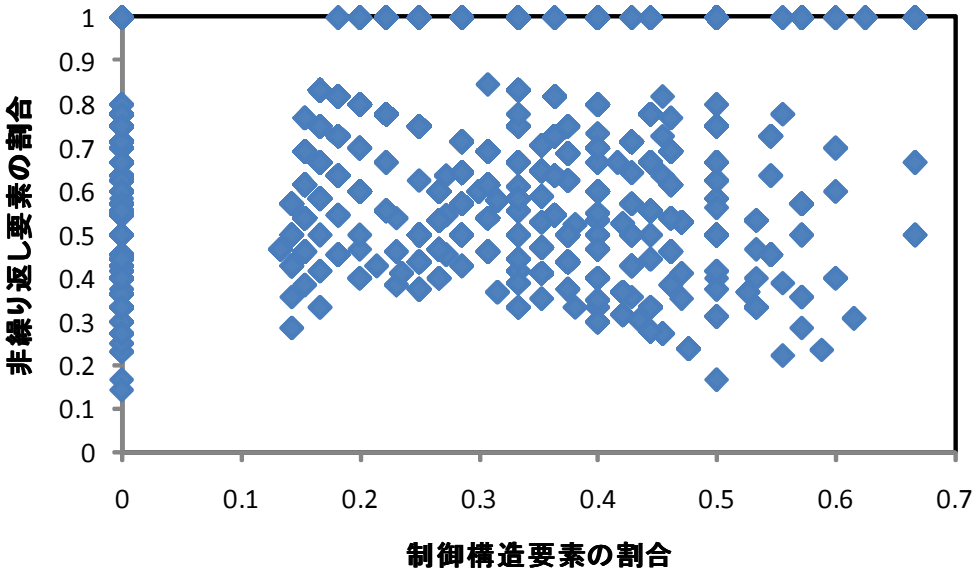

図 15: [Apache Tomcat] 制御構造要素の割合と非繰り返し要素の割合

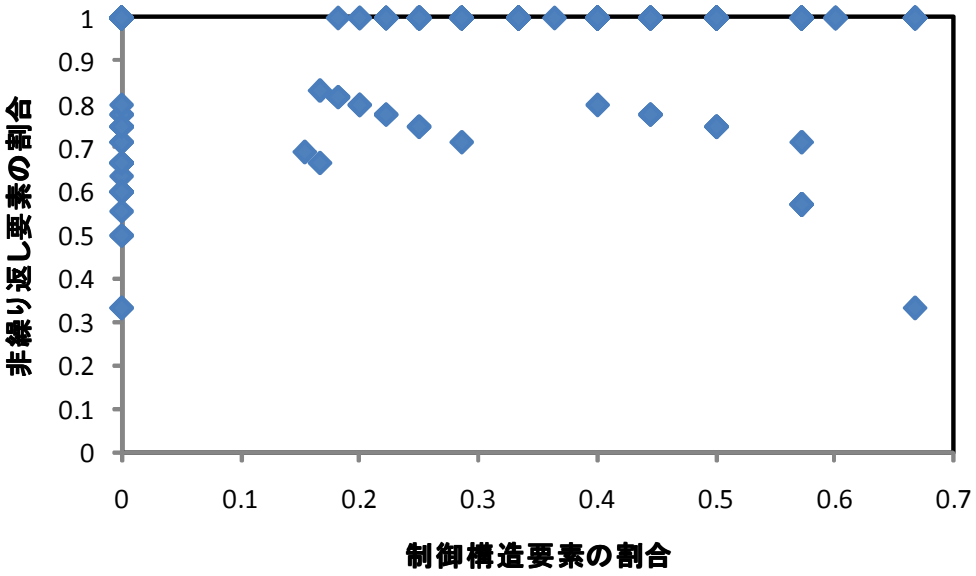

図 16: [SableCC] 制御構造要素の割合と非繰り返し要素の割合

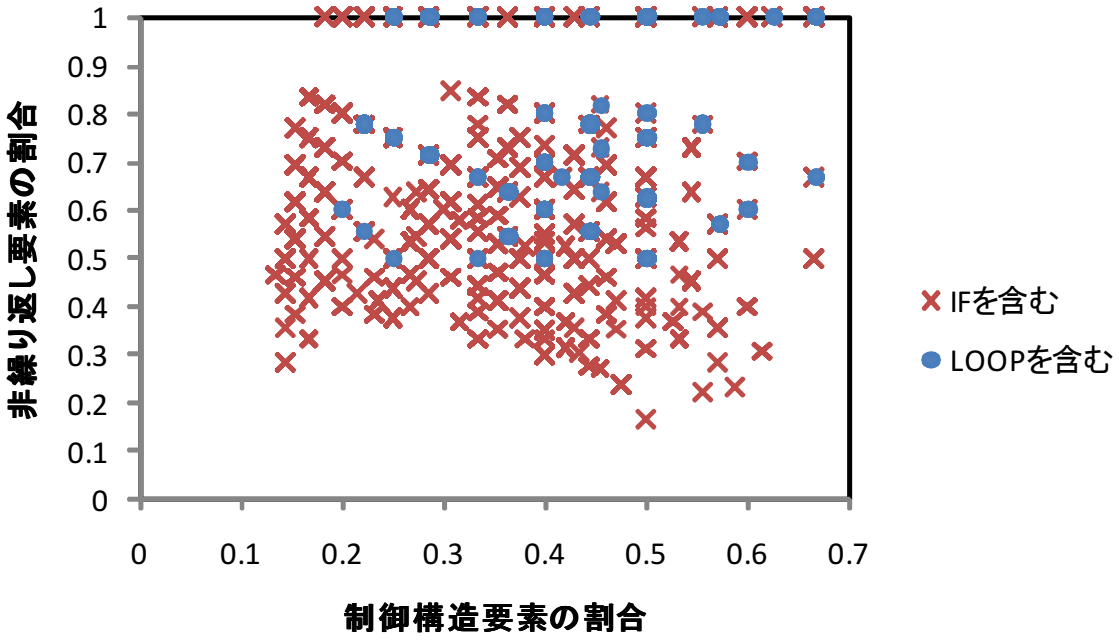

図 17: [Apache Tomcat] 制御構造要素の割合と非繰り返し要素の割合（IF，LOOP 分離版）

4.2.2 イテレータと出現位置の関連

本項では，イテレータの利用とパターンインスタンスの出現位置の関連性を分析する．パターンインスタンスの出現位置を表すメトリクスとしては，パターンインスタンスの分散を用いる．

イテレータの利用と出現位置の関連を分析するためには，コーディングパターンの中からイテレータの利用を含むパターンを選び出す必要がある．

イテレータの利用パターンの特定

イテレータの利用は，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を含んでいる．また，イテレータの利用では，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，次の条件を満たすコーディングパターンを，イテレータ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要素を持つ．
- メソッド名に「hasNext」を含むメソッド呼び出し要素を持つ．
- 「LOOP」,「END-LOOP」で表現された，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 3 に示す．

また，コーディングパターンをイテレータの利用パターンと，イテレータの利用に関係がないその他のパターンに分類し，図 18〜図 21 にパターンの分散とインスタンス数の関連を示した．

図 18，19 に示すように，JHotDraw と jEdit に含まれる，イテレータを利用しているコーディングパターンは，パターンの分散が大きい．また，インスタンス数が多い，イテレータを利用したパターンも発見されている．

表 3: イテレータパターンの数

ソフトウェア名	パターン総数	イテレータ利用パターン数	割合
JHotDraw	375	8	2.1%
jEdit	2902	28	0.9%
Apache Tomcat	8782	434	4.9%
SableCC	450	132	29.3%

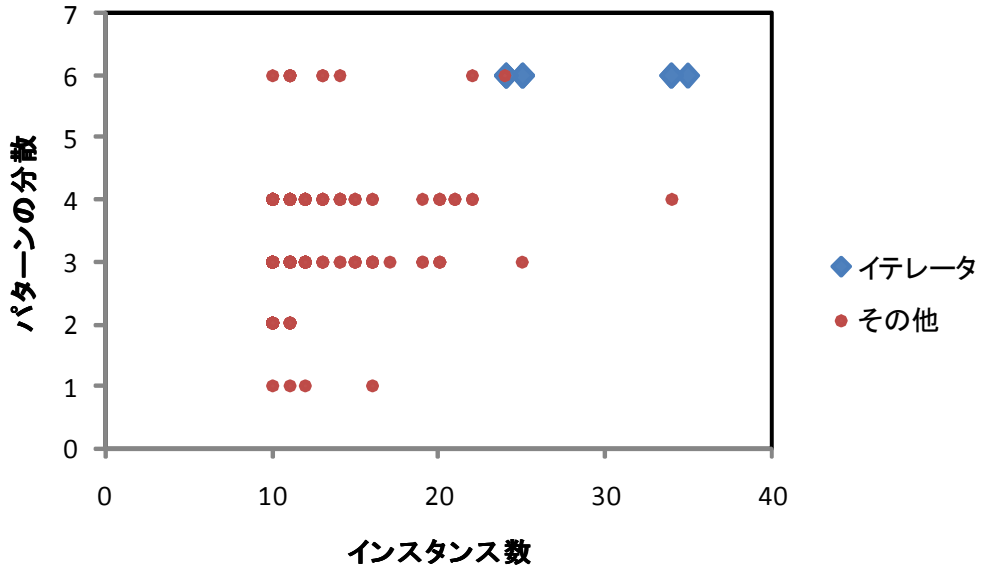

図 18: [JHotDraw] パターンの分散とインスタンス数の関連性

図 21 に示した SableCC の場合では，SableCC のパッケージ階層が浅く作られ 1 つのパッケージに多数のソースファイルが格納される構造になっていた．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 20 に示した Apache Tomcat では，パッケージ階層の深さは，分布を判断する上では十分であるが，傾向はみられなかった．これは，イテレータとプログラム固有の機能実装の両方を含んだパターンが，パターン階層中で局所的に現れていることが，原因として考えられる．

イテレータの利用は，Java のプログラマ間では既知の事項であり，パターンとしての重要度は低い．しかし，表 3 によると，SableCC から抽出されたパターンの中では，イテレータを含むパターンの割合は 29.3%に達している．

イテレータパターンを取り除く指標を作成することで，ユーザがプログラムを理解するために有用な，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．

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順に調査を行っていた．しかし，JHotDraw や jEdit のように，イテレータ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ていることから，有益なパターンが多数のコーディングパターン中に埋もれてしまうことも考えられる．この問題を回避するためには，パターンの分散が低いパターン，つまり，特定のパッケージやファイルに限定して登場しているパターンから順に調査するといった方法に切り替える必要がある．

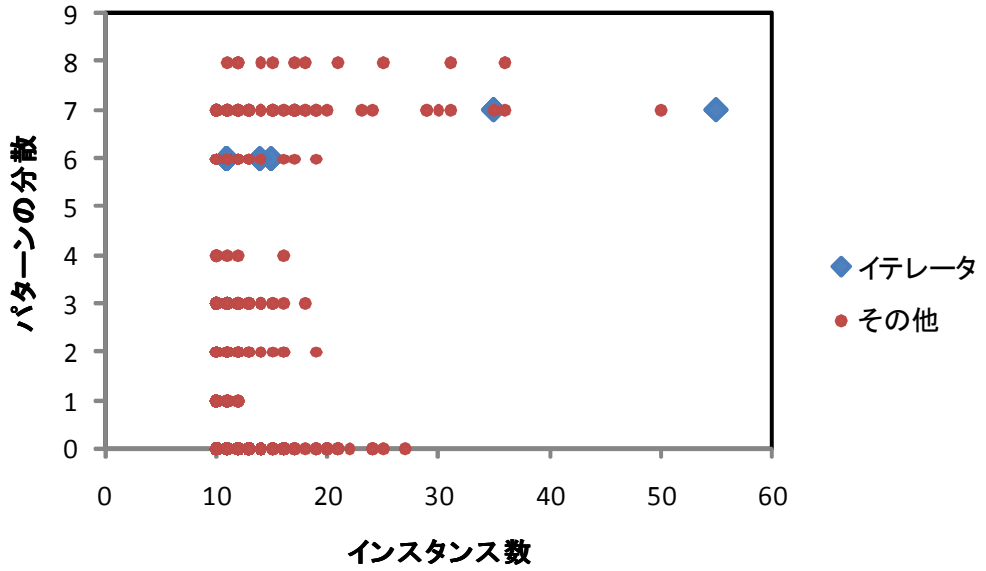

図 19: [jEdit] パターンの分散とインスタンス数の関連性

また，ソフトウェア全体のパッケージ階層の浅いソフトウェアに関しては，傾向の判断が困難であるため，パターンの出現位置の情報として利用するメトリクスを改良する必要がある．

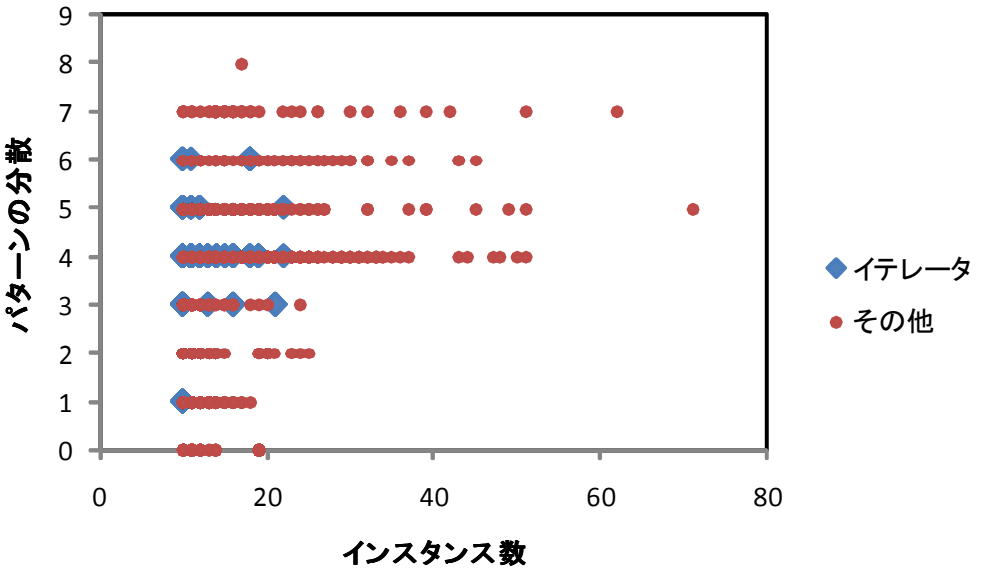

図 20: [Apache Tomcat] パターンの分散とインスタンス数の関連性

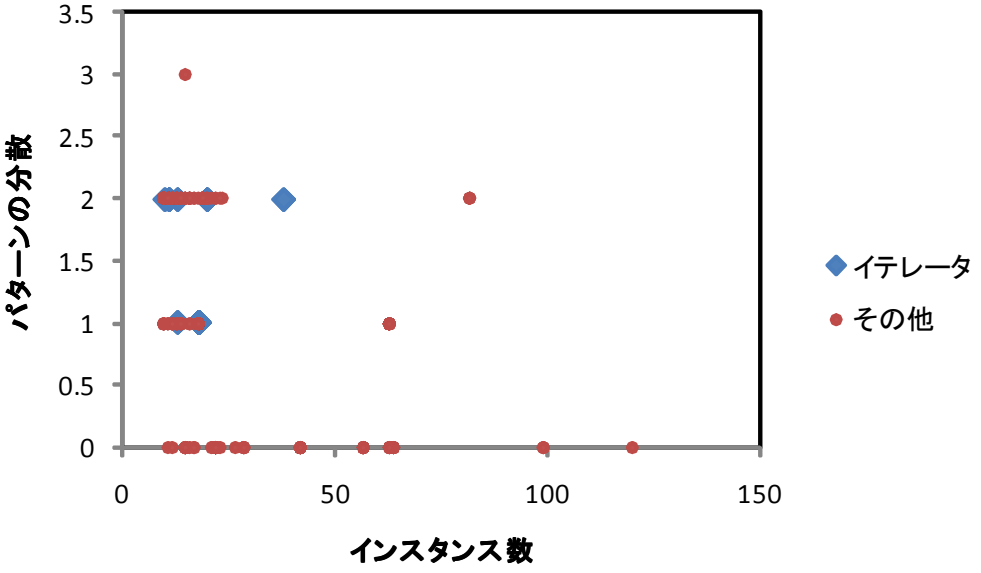

図 21: [SableCC] パターンの分散とインスタンス数の関連性

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，必要としている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は，コーディングパターンを種類別に分類し，コーディングパターン閲覧者が必要としているもののみ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などが，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．

今後，本研究の分析結果をもとに，コーディングパターンのフィルタリング手法の実現を目指す．また，コーディングパターンを効果的に開発者に提示するために，コーディングパターン閲覧環境を統合開発環境上で提供することや，コーディングパターンからソースコードの利用方法を抽出し，ソフトウェア部品検索システム上でソフトウェア部品と共に提供することなどが，発展研究としてあげられる．

謝辞

本研究の全過程を通して，常に適切な御指導および御助言を賜りました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 井上 克郎 教授に心より深く感謝いたします.

本研究を通して，随時適切な御指導および御助言を頂きました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 松下 誠 准教授に深く感謝いたします.

本研究を通して，適切な御指導および御助言を頂きました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 石尾 隆 助教に深く感謝いたします.

本研究に対して，適切な御指導および御助言を頂きました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 早瀬 康裕 特任助教に深く感謝いたします.

最後に，様々な御指導および御助言を頂きました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 ソフトウェア工学講座の皆様に感謝いたします.

参考文献

[1] Apache Tomcat. `http://tomcat.apache.org/`.

[2] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, Vol. 6, pp. 49–57, 1992.

[3] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[6] 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.

[7] Takashi Ishio, Hironori Date, Tatsuya Miyake, and Katsuro Inoue. Mining coding patterns to detect crosscutting concerns in java programs. In *Proceedings of the 15th IEEE Working Conference on Reverse Engineering*, pp. 123–132, 2008.

[8] jEdit. `http://www.jedit.org/`.

[9] JHotDraw. `http://www.jhotdraw.org/`.

[10] T. Kamiya, S. Kusumoto, and K. Inoue. 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.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp. 220–242, 1997.

[12] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proceedings of the 29th International Conference on Software Engineering*, pp. 106–115, 2007.

[13] M. Marin. Reasoning about assessing and improving the seed quality of a generative aspect mining technique. In *Proceedings of the 2nd International Linking Aspect Technology and Evolution Workshop*, 2006.

[14] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 132–141, 2004.

[15] Tatsuya Miyake, Takashi Ishio, Koji Taniguchi, and Katsuro Inoue. Towards maintenance support for idiom-based code using sequential pattern mining. In *Asian Workshop on Aspect-Oriented Software Development(AOASIA3)*, 2007.

[16] C.G. Nevill-Manning and I.H. Witten. Identifying hierarchical structure in sequences: A linear-time algorith. *Journal of Artificial Intelligence Research*, pp. 67–82, 1997.

[17] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of the 17th International Conference on Data Engineering*, pp. 215–224, 2001.

[18] SableCC. `http://sablecc.org/`.

[19] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 264–280, 2000.

[20] 中山崇, 松下誠, 井上克郎. ソースコードの差分を用いた関数呼び出しパターン抽出手法の提案. 情報処理学会研究報告，2006-SE-151, pp. 49–56, 2006.