特別研究報告

題目

オブジェクト指向プログラムに対するメソッド呼び出しパターン違反の検出手法

指導教員

井上 克郎 教授

報告者

山田 吾郎

平成 21 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成20年度 特別研究報告

オブジェクト指向プログラムに対するメソッド呼び出しパターン違反の検出手法

山田 吾郎

内容梗概

データマイニングの手法の1つとして, シーケンシャルパターンマイニングが挙げられる. これは, 順序を持つ要素の系列 (シーケンス) の集合から, 頻出する部分系列を抽出する手法である. シーケンシャルパターンマイニングをソースコード中のメソッド定義に含まれるメソッド呼び出しの系列に適用することで, メソッド呼び出しの部分系列のうち, 頻出するものが抽出される. これは, メソッド呼び出しパターンやコーディングパターンとよばれる. ソースコード中で, メソッド呼び出しパターンの部分系列が少数のみ見つかった場合, その部分系列がパターン違反である. パターン違反は, その出現位置で欠陥が生じている可能性が高いことが知られている.

しかし, 従来の研究で行われたパターン違反を用いた欠陥検出は, 手続き型言語であるC言語により記述されたプログラムを対象としており, 近年普及してきているオブジェクト指向プログラムへの適用事例は確認されていない.

本研究では, メソッド呼び出しパターンのパターン違反を用いた欠陥検出を, オブジェクト指向型言語であるJava言語で記述されたプログラムに適用した. 手続き型プログラムでは関数を関数名で識別できるが, オブジェクト指向プログラムではクラス階層があるため, メソッド名だけではメソッドを識別するには不十分である. そこで, メソッド名に加え, メソッドの引数の型などのメソッドに関連する型を用いてメソッドを識別する手法を提案する. 適用実験として, パターン違反を検出するツールを作成し, Java言語で記述されたオープンソースのプログラムから, 型を考慮する場合としない場合の2通りで検出を行った. その結果, 型を考慮して検出したパターン違反の中のみに欠陥が含まれていた.

主な用語

シーケンシャルパターンマイニング
欠陥検出
パターン違反
メソッド呼び出しパターン

目 次

1 まえがき **4**

2 背景 **5**

2.1 頻出パターンマイニング 5
2.2 パターン違反 5
2.3 パターン違反の検出手法 6
2.4 パターン違反を用いた欠陥検出 7
2.5 関連研究 7
2.5.1 頻出パターンマイニングを用いたソースコード解析 7
2.5.2 メソッド呼び出しパターン抽出ツール Fung 9
2.5.3 パターン違反を用いた欠陥検出事例 12
2.6 問題点 13
2.6.1 メソッド呼び出しパターンの誤認識による検出漏れ 13
2.6.2 サポート値の増加に伴う確信度の減少による検出漏れ 14

3 提案手法 **15**

3.1 処理 1. Fung を用いたメソッド呼び出しパターンの抽出 17
3.2 処理 2. メソッド呼び出しパターンの型による分類 19
3.3 処理 3. 分類されたメソッド呼び出しパターンのグループ化 20
3.4 処理 4. 相関ルールの確信度を用いたパターン違反検出 20

4 適用実験 **23**

4.1 実験目的 23
4.2 評価方法 23
4.3 実験対象 23
4.4 実験結果 24
4.5 考察 24
4.5.1 オブジェクト指向プログラムに対するメソッド呼び出しパターンのパターン違反を用いた欠陥検出の有効性の評価 24
4.5.2 型を考慮することの有効性の評価 26

5 まとめ **28**

謝辞 **29**

参考文献 30

1 まえがき

データマイニングの手法の1つとしてシーケンシャルパターンマイニング [2] (Sequential Pattern Mining) が挙げられる．これは, 順序を持つ要素の系列 (シーケンス) の集合から, 頻出する部分系列を抽出する手法である．

シーケンシャルパターンマイニングを用いることで, ソースコード中のメソッド定義から, 頻出するメソッド呼び出しの系列を抽出する研究が行われている [17][18]．このような頻出するメソッド呼び出しの系列は, メソッド呼び出しパターンやコーディングパターンと呼ばれ, 横断的関心事 [10] やイディオム [16] を表していると指摘されている [18]．

また, メソッド呼び出しパターンの中には, 実装を行う上でのルールを表しているものが含まれている．このことに着目して, メソッド呼び出しパターンに違反するメソッド呼び出しの系列を含むコード片を検出することにより, 欠陥の検出を行う研究が行われている [9][12]．具体的には, まずメソッド呼び出しの系列の出現数と比較して, 出現数が相対的に少ない部分系列をパターン違反として検出する．そして, パターン違反を含むコード片を欠陥を含むコード片の候補として提示する．

しかし, 従来の研究で行われたパターン違反を用いた欠陥検出は, 手続き型言語であるC言語により記述されたプログラムを対象としており, 近年普及してきているオブジェクト指向プログラムへの適用事例は確認できていない．

本研究では, メソッド呼び出しパターンのパターン違反を用いた欠陥検出を, オブジェクト指向型言語であるJava言語で記述されたプログラムに適用した．手続き型プログラムでは関数を関数名で識別できるが, オブジェクト指向プログラムではクラス階層があるため, メソッド名だけではメソッドを識別するには不十分である．そこで, メソッド名に加え, メソッドの引数の型などのメソッドに関連する型を用いてメソッドを識別するパターン違反検出手法を実現した．適用実験として, パターン違反を検出するツールを作成し, Java言語で記述されたオープンソースのプログラムから, 型を考慮する場合としない場合の2通りで検出を行った．その結果, 型を考慮して検出したパターン違反の中のみに欠陥が含まれていた．

以降, 2節ではパターン違反を用いた欠陥検出について, それに必要な概念の説明を行い, また, 研究の背景となる関連研究について述べる．3節では, 2節で述べた手法の詳細を説明する．4節では, 行った適用実験とその考察を行い, 最後に5節で本研究のまとめと今後の課題について述べる．

2　背景

本節では, 頻出パターンマイニングに基づいたパターン違反による欠陥検出に必要な概念を述べた後, 関連研究について述べる.

2.1　頻出パターンマイニング

頻出パターンマイニングとは, データマイニングの一種である [7]. データの集合をトランザクションとよび, トランザクションの集合に対しマイニングを行うことで, 一定回数以上出現するデータのセット (順序を持たない集合) やシーケンス (順序を持つ集合) などのパターンを得ることを目的とする. 頻出するセットを得るための手法をアイテムセットマイニングと言い [6], 頻出するシーケンスを得るための手法をシーケンシャルパターンマイニングと言う [2]. また, トランザクション中でパターンに該当する部分をパターンのインスタンスとよぶ. 逆に, インスタンスを抽出したものがパターンとも言える.

以下, 節 2.2, 2.3, 2.4 では, アイテムセットマイニングとシーケンシャルパターンマイニング両者に共通する, パターン違反を用いた欠陥検出の手法について説明するため, マイニングにより得るパターンの種類をシーケンスやセットと限定せず単にパターンと言うことにする.

2.2　パターン違反

パターン違反とは, 頻出パターンマイニングにより得られたパターンをルールとみなしたとき, そのルールに違反するインスタンスを指す. パターン違反は, それが存在するトランザクションがインスタンスの付近で異常をきたしている可能性を示す.

頻出パターンマイニングにより得られたパターンを P としたとき, P の部分列からなるパターン, すなわち, P より要素数が少なく, かつ全ての要素が P に含まれているパターン P' を, パターン P のサブパターンとよぶことにする. より厳密に次のように書くことができる.

P を次のように定義する.

$$P = \{e_1, e_2, \cdots e_n\} \tag{1}$$

このとき, そのサブパターンとは, 次の条件, $1 \leq m < n$ かつ $1 \leq s_k \leq n (1 \leq k \leq m)$ かつ $s_k < s_{k+1} (1 \leq k \leq m-1)$ において, 以下の数式で定義される集合および系列である.

$$P' = \{e_{s_1}, e_{s_2}, \cdots e_{s_m}\} \tag{2}$$

パターン P が出現するトランザクションの数を, パターン P のサポート値と言い, それを $S(P)$ と表記する. 以下の性質が成り立つ.

パターンQが出現するトランザクションの集合をT(Q)と表記する．

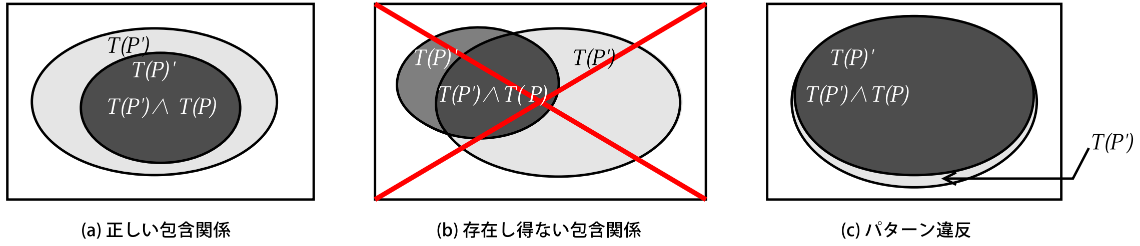


図 1: パターン P とそのサブパターン P' の包含関係を表した図

- パターン P が出現するトランザクションには, 必ず P' も現れる.
- $S(P') \geq S(P)$

ベン図による例を図 1 に示す. 図中の (a) はパターン P がサブパターン P' に包含されている正しい包含関係を表している. 一方, (b) は P の一部分が P' と共通集合を持っておらず, このようなことは起こらない.

図 1(c) のように, あるパターン P の出現するトランザクションと P のサブパターン P' の出現するトランザクションとの差集合が極めて小さいとき, サブパターン P' を P の変種と言い, 差集合のトランザクション中で現れる P の変種のインスタンスをパターン違反と言う. 違反とみなす差集合の大きさは任意である.

例えば, パターン $P = \{e_1, e_2, e_3\}$ が 99 箇所のトランザクションに現れたのに対し, サブパターン $P' = \{e_1, e_3\}$ が 100 箇所のトランザクションで出現した場合, P' のうち 99 箇所は P も出現しており, P' のみ出現するトランザクション, すなわち P と P' が出現するトランザクションの差集合は 1 箇所となる. P の出現数 99 に比べ, P' との差集合である 1 がパターン違反たりうる小ささであるとみなせるとき, P' は P の変種であり, 差集合のトランザクションに出現する P' のインスタンスをパターン違反として検出する.

2.3 パターン違反の検出手法

パターン違反は相関ルール [1] の確信度を用いて検出する.

相関ルールとは, トランザクション T において, パターン P_1 が存在したとき, パターン P_2 も存在すると言う規則で, $P_1 \Rightarrow P_2$ と書く. 相関ルールが成立する強さを確信度と言い, P_1 が出現したトランザクションで P_2 も出現する条件付き確率で計算される. これは, 前節 2.2 でのサポート値の定義を用いて次のように表すことができる.

$$確信度 = \frac{S(P_1 \bigcup P_2)}{S(P_1)} \tag{3}$$

したがって確信度は 0.0 ∼ 1.0 の間で表される.

相関ルールの確信度が 1.0 ではない十分大きな値のとき, $P_1 \Rightarrow P_2$ の左辺, P_1 を変種とし, P_1 が出現するトランザクションと P_2 の出現するトランザクションの差集合中での変種のインスタンスをパターン違反とする. 大きいとみなす値は任意の閾値で決定される. ここで確信度が 1.0 の相関ルールを除いているのは, 確信度が 1.0 と言うのは全ての P_1 が出現するトランザクションで P_2 が出現しており, パターン違反が起こっていないからである.

2.4 パターン違反を用いた欠陥検出

ソースコードのメソッド定義中で呼び出されているメソッド名をトランザクションとし, 全てのメソッド定義から生成したトランザクションに対し頻出パターンマイニングを行う. この時得られたパターンのパターン違反は, メソッド呼び出しの欠落などによる欠陥である可能性が高い.

図 2 に模擬コードによる例を示す.

図 2(a) では, パターン $\{function1, function2, function3\}$ がトランザクション $\{functionA\}$, $\{functionD\}$ の計 2 箇所で見つかっており, そのサブパターン $\{function1, function3\}$ のみが $\{functionB\}$, $\{functionC\}$, $\{functionX\}$ の 3 箇所で出現している. $\{function1\}$, $\{function3\}$ のみを使う場合もあれば, その間に $\{function2\}$ を呼び出す用例もあると考えることができる.

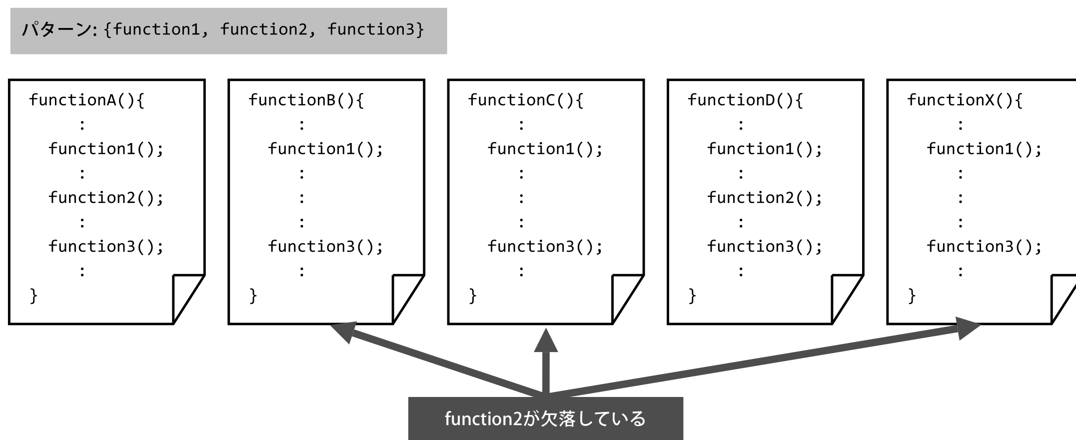
一方, 図 2(b) では, パターン $\{function1, function2, function3\}$ がトランザクション $\{functionA\}$, $\{functionB\}$, $\{functionC\}$, $\{functionD\}$ の計 4 箇所で見つかっているのに対し, そのサブパターン $\{function1, function3\}$ のみが出現するトランザクションは $\{functionX\}$ の 1 箇所である. この場合 $\{functionX\}$ でのサブパターンは $\{function2\}$ の欠落による欠陥が生じている可能性が高いと考えることができる.

2.5 関連研究

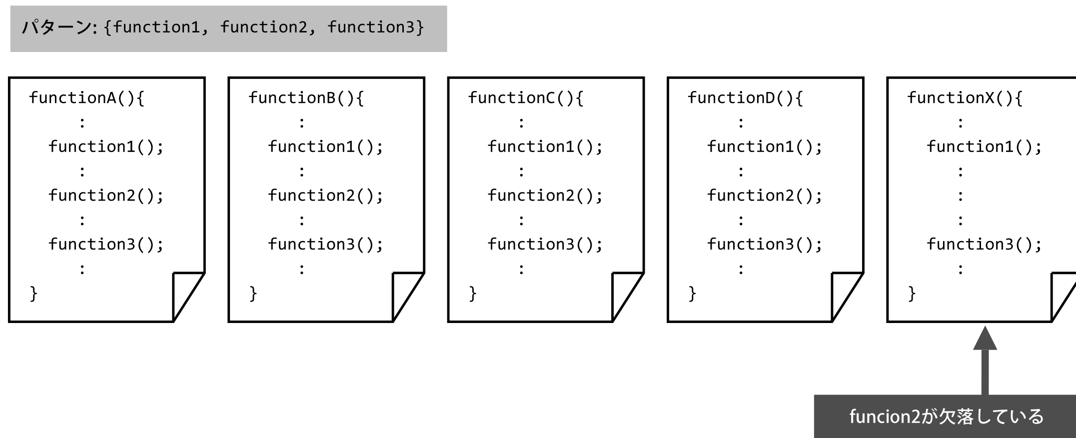
2.5.1 頻出パターンマイニングを用いたソースコード解析

Michail[13] は, データマイニングの手法の 1 つである相関ルールマイニングを用い, プログラムライブラリを使用したソースコードから, ライブラリ関数の使用方法を自動抽出する手法を提案した. この手法により, ライブラリ関数の理解支援に役立つパターンが抽出できた.

中山ら [17] は, メソッド呼び出しパターンを用いた理解支援において, 1 つのパターン内に異なる複数の機能が混在してしまう問題の解決について研究を行った. この研究では, ソースコードは機能単位で変更されるという仮説の下, ソースコードの変更差分に対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, その結果, 単一の機能のみ含むメソッ



(a) 欠陥である可能性が低い場合



(b) 欠陥である可能性が高い場合

図 2: 模擬コードによるパターン違反の例

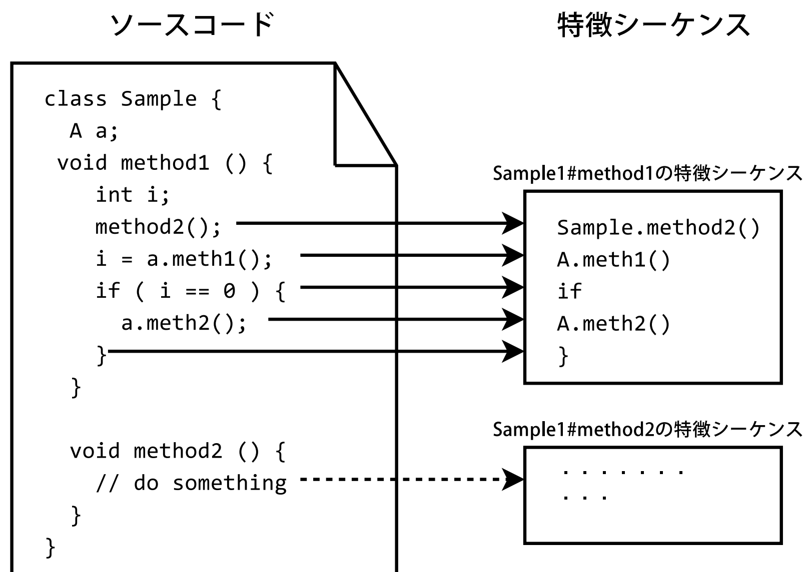


図 3: 特徴シーケンス

ド呼び出しパターンの抽出に成功した.

また, Li ら [11] はシーケンシャルパターンマイニングを用いて欠陥検出を行った. まず, シーケンシャルパターンマイニングによりコピーアンドペーストから生成されたコード片の自動抽出を行い, 次に抽出されたコード片について, 識別子の変更率を基準に欠陥の候補を挙げる. この手法を C 言語で記述された Linux カーネルなどに適用し, Linux カーネルからは 49 の欠陥を発見した.

頻出パターンマイニング抽出ツールはいくつがあるが, 次の節 2.5.2 でそのうちの 1 つである Fung について説明する.

2.5.2 メソッド呼び出しパターン抽出ツール Fung

Fung[14] は Java 言語を対象としたシーケンシャルパターンマイニングの抽出ツールである. Fung はソースコード中のメソッド, つまり関数の定義から, メソッド呼び出しのメソッド名と制御構造, すなわち if 文などの条件文や while 文などの繰り返し文の開始・終了位置からなる特徴シーケンスをトランザクションとし, 特徴シーケンスの集合に対しマイニングを行う.

Fung では, シーケンシャルパターンマイニングの実装に PrefixSpan アルゴリズム [15] を用いている. ソースコード中の全てのメソッド定義から特徴シーケンスを作成したのち (図 3), それらの特徴シーケンスに対して PrefixSpan アルゴリズムによるシーケンシャルパターンマイニングを行う. 以下で PrefixSpan アルゴリズムの概略を示す.

手順 1. それぞれの特徴シーケンスを構成している全ての要素のサポート値を計算する.

手順 2. 任意に設定した最小サポート値を越える要素を, シーケンシャルパターンとして出力する.

手順 3. 閾値を越える各要素について射影を行う. 射影とは, 全てのシーケンスから特定の要素に続く接尾辞を取り出す操作である.

手順 4. 手順 3 の射影により得られた特徴シーケンスに対し, 再び手順 1 のからの操作を繰り返す.

図 4 に PrefixSpan アルゴリズムの例を示した. この例では, 最小サポート値を 2 として, 次の 4 つの特徴シーケンスを対象に PrefixSpan アルゴリズムを適用した.

特徴シーケンス 1. $\{method1 \rightarrow method2 \rightarrow method3\}$

特徴シーケンス 2. $\{method1 \rightarrow method3 \rightarrow method4\}$

特徴シーケンス 3. $\{method3 \rightarrow method2 \rightarrow method1\}$

特徴シーケンス 4. $\{method1 \rightarrow method1 \rightarrow method1\}$

図 4 中に記した第 1 段階について, 先ほど記載した手順 2〜4 になぞらえて説明する.

まず手順 2 では, 全ての特徴シーケンスから, 出現する要素の集合を作成し, それぞれの要素のサポート値, すなわち, 出現する特徴シーケンスの数を計算する. 集合は次のようになる.

$\{method1, mehtod2, method3, method4\}$

サポート値の計算方法について $method1$ を例に説明する. $method1$ は 4 つ全ての特徴シーケンスに出現するためサポート値は 4 となる. 特徴シーケンス 4 では $method1$ が 2 つ出現しているが, 数え上げるのは出現する特徴シーケンスの数であるためサポート値が 5 とはならない. 全ての要素についてサポート値の計算が終れば手順 3 に進む.

次に, 手順 3 では最小サポート値を上回る要素をメソッド呼び出しパターンとして出力する. ここでは最小サポート値を 2 としているため出力は次の 3 つとなる.

- $\{method1\}$
- $\{method2\}$
- $\{method3\}$

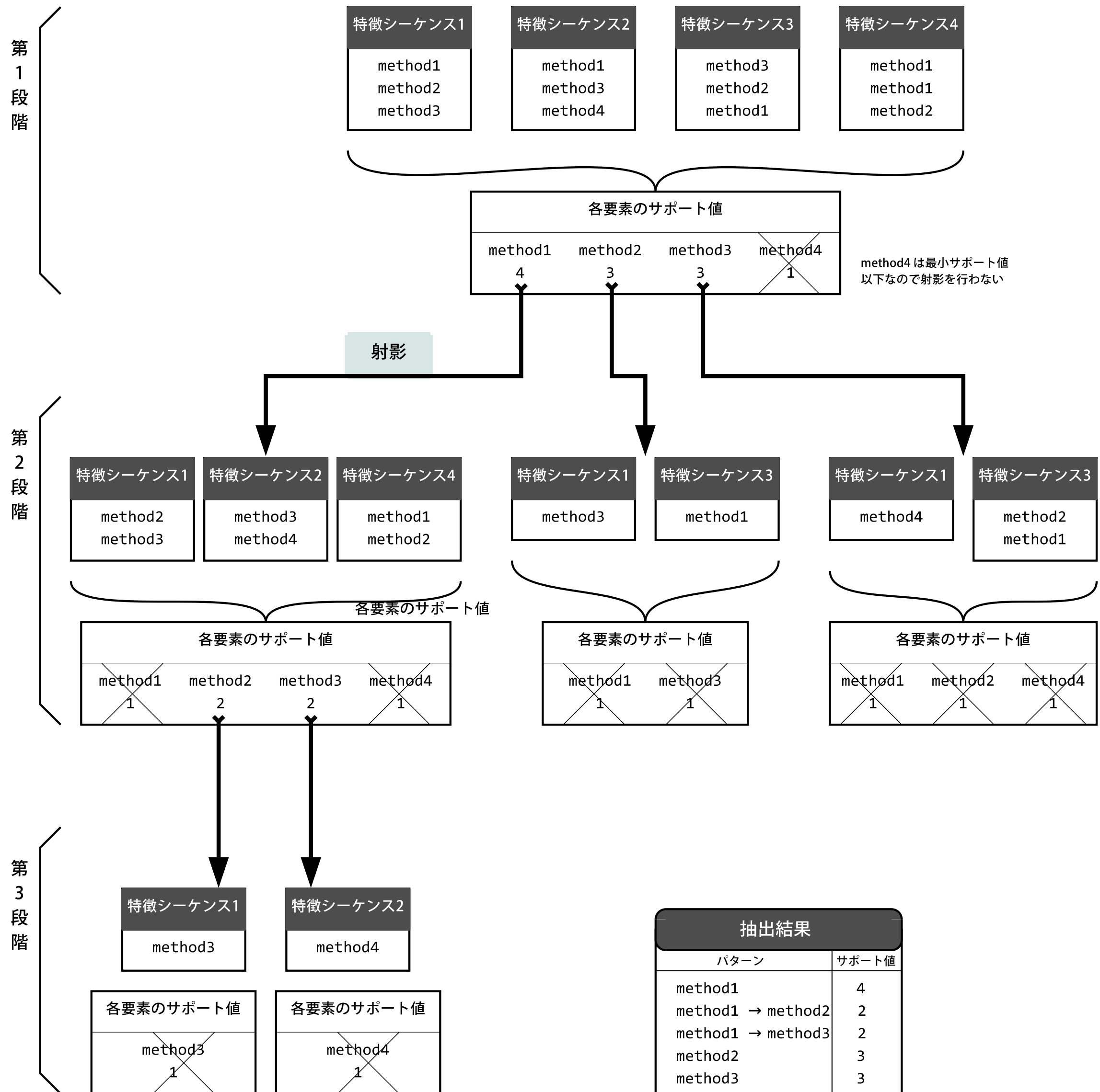


図 4: 最小サポート値を 2 としたときの PrefixSpan アルゴリズム

第 1 段階では 1 つの要素をもつメソッド呼び出しパターンのみ出力されるが, 第 2 段階以降であればそれまでに射影を行ってきた要素も出力する．例えば, 第 2 段階で $method1$ から射影された特徴シーケンスについて考える．ここでは $method2$, $method3$ が最小サポート値を上回っているため出力の対象となる．また, これらは $method1$ から射影された特徴シーケンスの中の要素であるため, 出力されるメソッド呼び出しパターンは先頭にそれまで射影を行ってきた要素, この場合は $method1$ を付加して次の 2 つとなる．

- $\{method1 \rightarrow method2\}$
- $\{method1 \rightarrow method3\}$

そして, 手順 4 で先ほど出力された 3 つの特徴シーケンスについて射影を行う．$method2$ を例にとると, これが出現する 3 つの特徴シーケンス, 特徴シーケンス 1, 3, 4 について射影を行い, $method2$ に続く特徴シーケンスを得る．特徴シーケンス 4 については, $method2$ の後に要素がないため, 射影は行われない．したがって, $method2$ に射影を行うことで得られる特徴シーケンスは 2 つとなっている．

第 1 段階以降, 射影する要素がなくなるまで同じ手順を繰り返す．その結果として次の 5 つの頻出シーケンスを抽出する．

- $\{method1\}$
- $\{method1 \rightarrow method2\}$
- $\{method1 \rightarrow method3\}$
- $\{method2\}$
- $\{method3\}$

実際にはあまり短いパターンには意味がないことが多いため, 閾値を設けてフィルタリングする．

2.5.3 パターン違反を用いた欠陥検出事例

Li ら [12] はアイテムセットマイニングを C 言語により記述されている Linux カーネルに対し適用し, 得られたアイテムセットのパターン違反から欠陥を検出し, アイテムセットマイニングに基づいたパターン違反による欠陥検出の有効性を示した．

Kagdi ら [9] は, Linux カーネルなど, C 言語で記述された複数の大規模プログラムに対しアイテムセットマイニングとシーケンシャルパターンマイニングをそれぞれ適用し, 欠陥検出と言う観点で有意なパターン違反の検出精度を比較した．その結果, シーケンシャルパターンマイニングの方がパターン違反の検出精度において優れていることが明かになった．

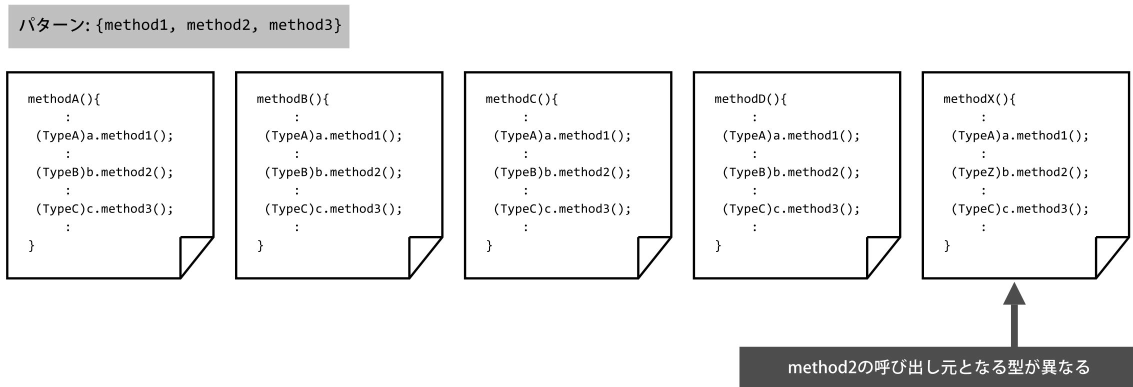


図 5: パターン違反であるべき部分をメソッド呼び出しパターンを誤認識してしまうことによる検出漏れ

2.6 問題点

オブジェクト指向プログラムでのパターン違反検出にあたって, 手続き型言語にはなかった問題が生じる. PrefixSpan アルゴリズムにおいて, サポート値とは同じメソッドが出現するメソッド定義の数である. そのためには, メソッド定義中の情報からメソッドの識別を行い, 同一のメソッドを特定する必要がある. 手続き型言語では, メソッド (関数) の特定はメソッド名のみで十分である. 一方, オブジェクト指向型言語では, クラスが異なれば同名のメソッドを定義することが可能であるため, メソッド名のみでメソッドを識別することができないという問題が起こる. さらに, 同名であるが異なるメソッドがパターンに含まれてしまうため, サポート値が増加してしまうという問題も起こる. これらは, 節 2.6.1, 2.6.2 で述べる検出漏れにつながる.

2.6.1 メソッド呼び出しパターンの誤認識による検出漏れ

まず考えられるのが, メソッド名のみ考慮して抽出したメソッド呼び出しパターンが, 実際は同名の異なるメソッドを呼び出しており, 欠陥であるのにもかかわらず検出できない場合である.

図 5 を例に考える. メソッド定義 $\{methodA\}$, $\{methodB\}$, $\{methodC\}$, $\{methodD\}$, $\{methodX\}$ において, パターン $\{method1, method2, method3\}$ が抽出されたが, $\{methodX\}$ のみ $\{TypeZ\}$ と言う型のオブジェクトを通して $\{method2\}$ を呼び出している. $\{methodX\}$ 中の $\{method2\}$ は, 他の $\{TypeB\}$ を通して呼び出されている $\{method2\}$ とは異なる可能性があり, パターン違反として検出されるべきである. しかし従来の手法ではマイニングの段階で型情報が欠落し, またパターン違反の段階でも考慮しないため, 検出漏れとなってしまう.

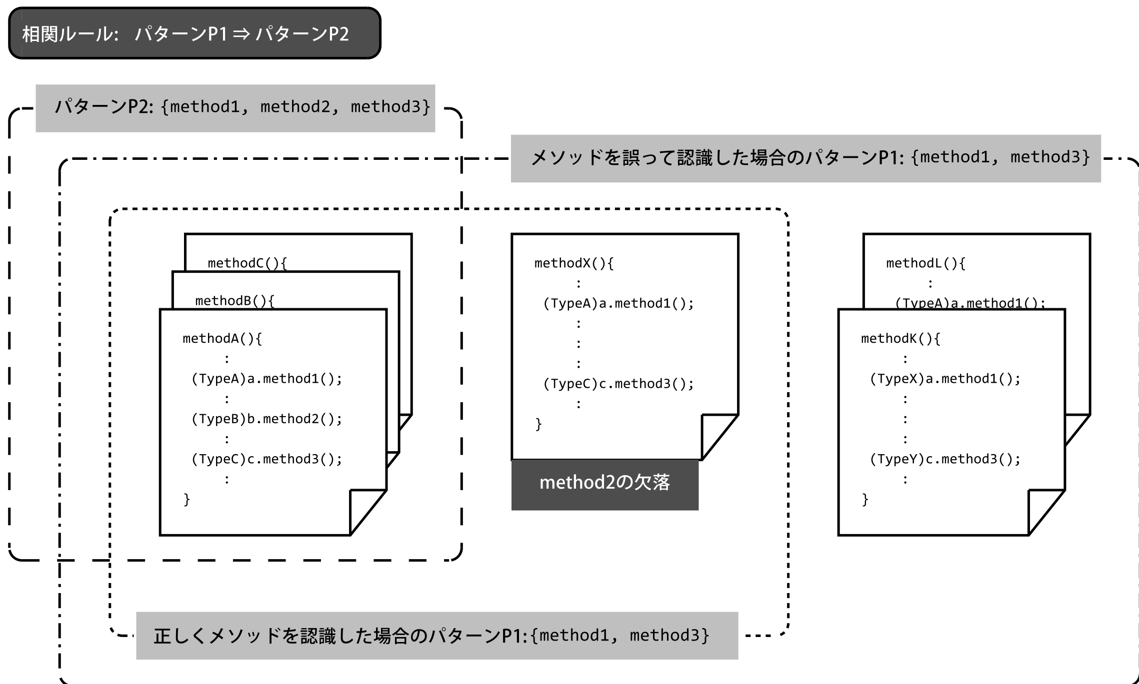


図 6: サポート値の増加に伴う確信度の減少による検出漏れの例

2.6.2 サポート値の増加に伴う確信度の減少による検出漏れ

次に考えられるのが, パターンの誤検出のために相関ルールの左辺のサポート値が増加し, それに伴い確信度が低下してしまい, パターン違反とみなす閾値に満たず検出漏れとなってしまう場合である.

図 6 に例を挙げた. パターン違反とみなす閾値を 0.7 としたとき, 2 つのパターン $P1, P2$ の相関ルール $P1 \Rightarrow P2$ についてのパターン違反を考える.

パターン $P1$ は, 正しくメソッドを識別できた場合, $methodA, methodB, methodC, methodX$ の 4 箇所で出現することになる. 一方, パターン $P2$ は, $methodA, methodB, methodC$ の 3 箇所で出現しており, 相関ルールの確信度は 式 3 より 0.75 となる. これは閾値より大きいため, パターン $P1$ のメソッド定義 $methodX$ 中におけるインスタンスはパターン違反とみなされる.

しかし, メソッド名のみ考慮する手法では, 先ほどの 4 つのメソッド定義に加え, 実際は異なるメソッドを呼び出している 2 つのメソッド定義, $methodK, methodL$ も含まれてしまう. この増加に伴い, パターン P1 のサポート値が増加し, 相関ルールの確信度は 0.5 となる. 結果, 閾値を下回ってしまいパターン違反として検出されない.

```
SomeClass.someMethod(parameter1, parameter2);
```

レシーバオブジェクト　抽出されたパターンを構成するメソッド　引数　引数

図 7: パターン違反検出の際に考慮する型

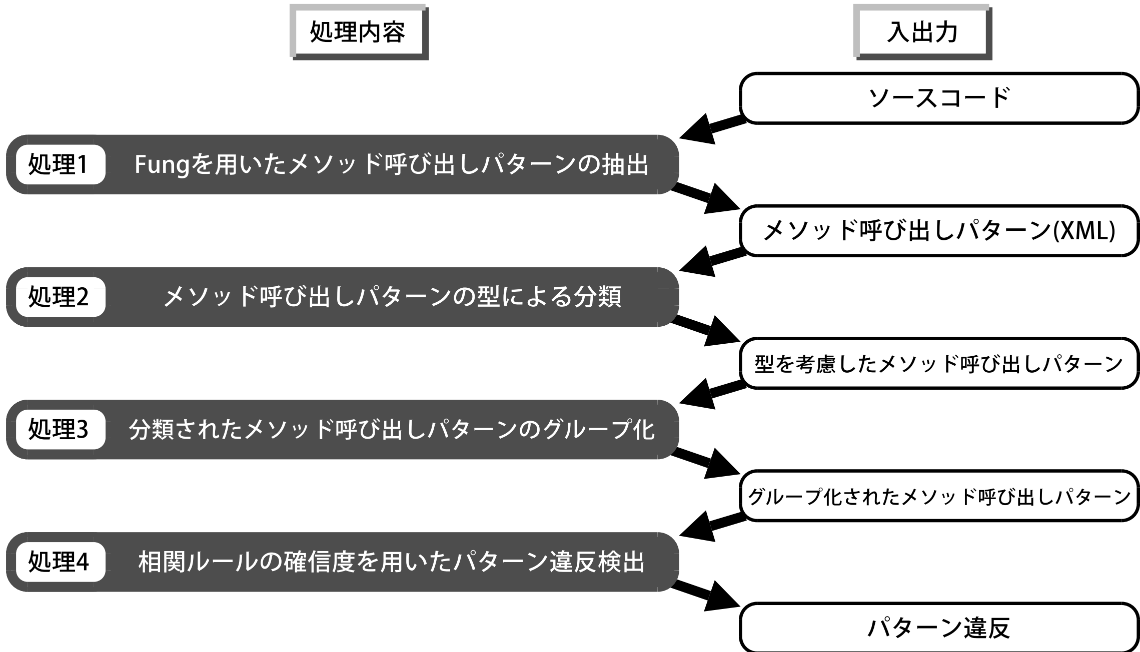


図 8: 提案手法の概要

3　提案手法

本研究では, パターン違反検出の際, メソッド呼び出しのメソッド名に加えメソッドに関連する型も利用することで, 前節 2.6 で述べたメソッドを特定できない問題の解決をはかる. 考慮する型とは以下の 2 つである.

- 呼び出されているメソッドのレシーバオブジェクトの型
- 呼び出されているメソッドの引数の型

図 7 に示すように, レシーバオブジェクトとはメソッド呼び出しを行っているオブジェクトである.

図 8 に本手法を用いたパターン違反検出の概要を示す. 以下でそれぞれの処理の簡単な説明を行う.

処理 1. Fung を用いたメソッド呼び出しパターンの抽出 まず, 対象プログラムのソースコー

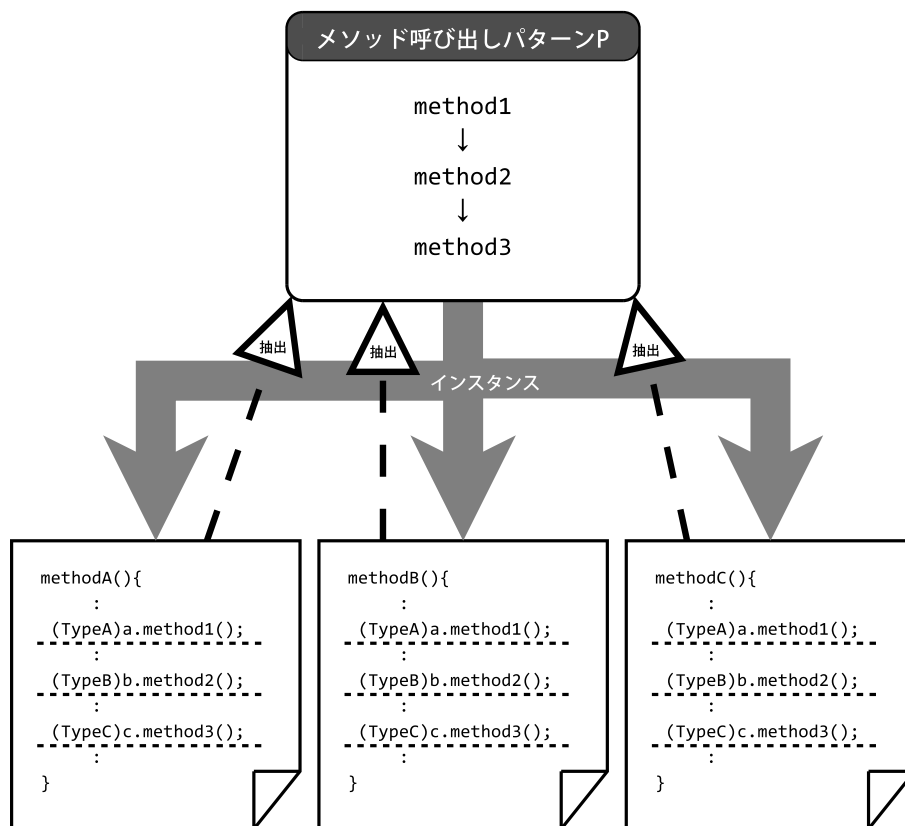


図 9: パターンとインスタンスの関係

ドを入力に Fung を実行する．抽出したメソッド呼び出しパターンは, 型情報とともに XML ファイルに出力される.

処理 2．メソッド呼び出しパターンの型による分類 次に, Fung から得たメソッド呼び出しパターンを, 型情報を用いてインスタンス毎に分類する．これにより型を考慮したメソッド呼び出しパターンを得る．インスタンスとは, ソースコード中でメソッド呼び出しパターンに対応する要素である．メソッド呼び出しパターンとインスタンスの関係は図 9 に示した．この図では, メソッド呼び出しパターン $P = \{method1, method2, method3\}$ があり, そのインスタンスが, ソースコード中のメソッド定義 $methodA, methodB, methodC$ に現れている.

処理 3．分類されたメソッド呼び出しパターンのグループ化 さらに, 相関ルールを作成するために, 型を考慮したメソッド呼び出しパターンをグループ化する．グループとは, メソッド呼び出しパターンとそのサブパターンからなる集合である.

処理 4．相関ルールの確信度を用いたパターン違反検出 最後に, それぞれグループについて, グループ内の型を考慮したメソッド呼び出 しパターン間における相関ルールの確信度を計測し, パターン違反を検出する.

以降, 節 3.1, 3.2, 3.3, 3.4 で手法の詳細を説明する.

3.1 処理 1．Fung を用いたメソッド呼び出しパターンの抽出

節 2.5.2 で述べたツール Fung を用い, メソッド呼び出しパターンの抽出を行う．ただし, Fung には若干の修正を加えており, 従来の出力に加えメソッドに関連する型の情報も出力する．メソッド呼び出しパターンを表す XML ファイルは大まかに次のような木構造をなす. なお, 末尾に $*$ と記した要素はその要素が 0 回以上出現することを示し, $+$ は 1 回以上の出現を意味する.

- メソッド呼び出しパターン $+$
  - インスタンス $+$
    - インスタンスの出現ファイル
    - インスタンスの出現クラス (オーナークラス)
    - インスタンスの出現メソッド (オーナーメソッド)
    - インスタンスの要素 $+$
      - メソッド名および制御構造の種類 (条件文, 繰り返し文)

```
<pattern-list length="22">
  <pattern number-of-instances="15" id="13" pattern-length="3">
    <pattern-name name="Method1" />
    <pattern-name name="Method2" />
    <pattern-name name="Method3" />
    <pattern-instance-list>
      <pattern-instance owner-class="ClassA" fileID="13" owner-method="MethodA">
        <sequence name="Method1" receiver-type="TypeA" ...>
          <parameter type="TypeB" />
          <parameter type="TypeC" />
          <parameter type="TypeD" />
          <begin column="13" line="50" />
          <end column="22" line="50" />
        </sequence>
        </sequence>
        <sequence name="next"  receiver-type="TypeB" ...>
        ...
      </pattern-instance>
      <pattern-instance ...
      ...
    </pattern-instance-list>
  </pattern>
  <pattern ...
  ...
</pattern-list>
```


図 10: メソッド呼び出しパターン抽出ツール Fung の出力例

· ソースコード中でインスタンスの要素が出現する行と列

制御構造の種類は次の 2 種類である.

- ループ
- 分岐

さらに, インスタンスの要素がメソッド呼び出しであった場合, 上記の 2 つの子要素に加えメソッド呼び出しに関連する型情報も持つ.

- 型情報 (メソッドのみ)
  - レシーバオブジェクトの型
  - 引数の型 ∗

図 10 に, 出力された XML の例を挙げる. XML 中で, 全てのメソッド呼び出しパターンは `pattern-list` の子要素 `pattern` として定義される. パターンのインスタンスは `pattern` の

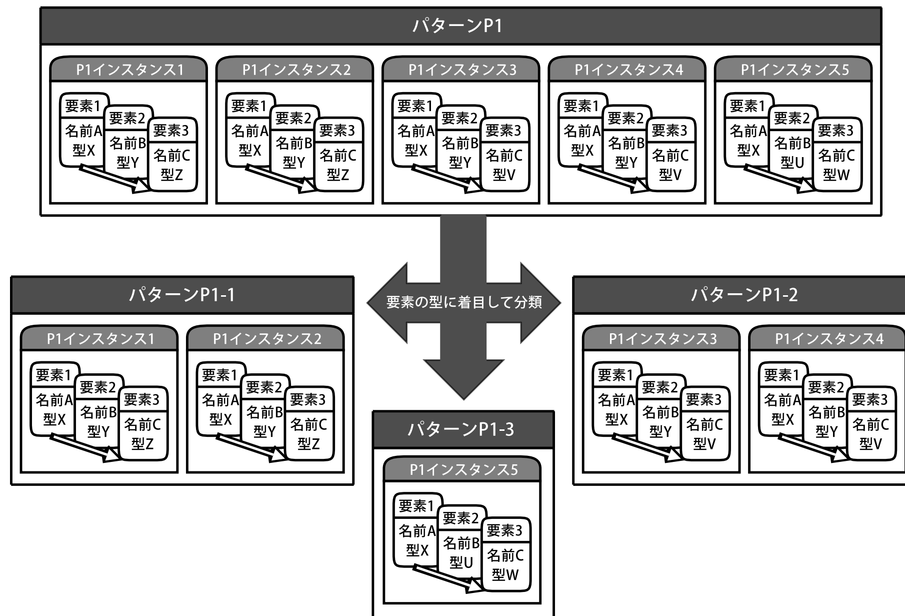


図 11: 型によるメソッド呼び出しパターンの分類

子要素 pattern-instance で列挙され, 各々のインスタンスの要素 (メソッド呼び出しおよび制御構造) が sequence 要素として定義される.

3.2 処理 2. メソッド呼び出しパターンの型による分類

節 3.1 で出力したメソッド呼び出しパターンを, XML に含まれるメソッド呼び出しに関連する型情報を用いて分類を行う. メソッド名のみ考慮して抽出したメソッド呼び出しパターンを, 型を考慮して分類を行うことで, メソッドの識別をより厳密に行う.

型による分類の例を図 11 に示した. メソッド呼び出しパターン P1 は, メソッド名 A, B, C のメソッド呼び出しで構成されている. パターン P1 は 5 つのインスタンス, P1 インスタンス 1, P1 インスタンス 2, P1 インスタンス 3, P1 インスタンス 4, P1 インスタンス 5 からなる. それぞれ, メソッド A, B, C の型 (ここではどの型なのか限定はしない) は,

1. **X, Y, Z:** P1 インスタンス 1, P1 インスタンス 2
2. **X, Y, V:** P1 インスタンス 3, P1 インスタンス 4
3. **X, U, W:** P1 インスタンス 5

となっている. これらはそれぞれ, パターン P1-1, パターン P1-2, パターン P1-3 と言う型を考慮したメソッド呼び出しパターンへと細分化される.

3.3 処理 3. 分類されたメソッド呼び出しパターンのグループ化

細分化された, 型を考慮するメソッド呼び出しパターンをグループ化する. グループとは, あるパターンと, そのサブパターンからなるメソッド呼び出しパターンの集合である.

グループ化は確信度の低い相関ルールを作成しないために行う. と言うのも, 2.3 で述べたように, パターン違反を検出するためには相関ルールの確信度を用いる. そのとき, どの 2 つのメソッド呼び出しパターンを相関ルールとするか決定する必要がある. 確信度が低い場合は閾値によりパターン違反とみなされない. また, 確信度は 2 つのメソッド呼び出しが同じメソッド定義で出現する割合と言える. したがって, 同じメソッド定義で現れると保証されているパターン同士の確信度を作成することが好ましい. サブパターンには, その元となるパターンの出現するメソッド定義に必ず出現する性質を持つため, グループ内のメソッド呼び出しパターン間で作成した相関ルールの確信度が低いとは考えづらい. そのため, グループ内で相関ルールを作成することで, 低い確信度になると考えられる相関ルールの作成を避けることができる.

グループ化の例を図 12 に挙げた. メソッド呼び出しパターン P1 を型で分類したパターン P1-1, パターン P1-2. メソッド呼び出しパターン P2 を分類したパターン P2-1, パターン P2-2. メソッド呼び出しパターン P3 を分類したパターン P3-1. この計 5 つのパターンが, それぞれパターン P1-1 とそのサブパターンで構成されるグループ 1, パターン P1-2 とそのサブパターンで構成されるグループ 2 にグループ化された. パターン P3-1 が双方に含まれていることに注意してほしい. 型で分類したパターンが属するグループは 1 つとは限定されない.

3.4 処理 4. 相関ルールの確信度を用いたパターン違反検出

グループ内に含まれる全てのメソッド呼び出しパターンについて相関ルールを生成し, その確信度を求める. 確信度が閾値以上であり 1.0 でなければパターン違反とみなす.

相関ルールは 2.3 で説明したように, あるメソッド呼び出しパターン P1 が存在するメソッド定義において, メソッド P2 も同時に出現すると言う規則であり, 次の式のように書く.

$$\{P1\} \Rightarrow \{P2\} \tag{4}$$

相関ルールの確信度は, 式 3 で定義したように, 式 4 における左辺のメソッド呼び出しパターンのサポート値 (メソッド定義の数) で, 両辺のメソッド呼び出しパターンが双方出現するメソッド定義の数を割った値である.

この規則が常に成り立つとき, すなわち確信度が 1.0 の場合は, その規則がパターン違反を含んでおらず, 欠陥検出の用途では不要である. また, 規則が多くのメソッド定義で成り

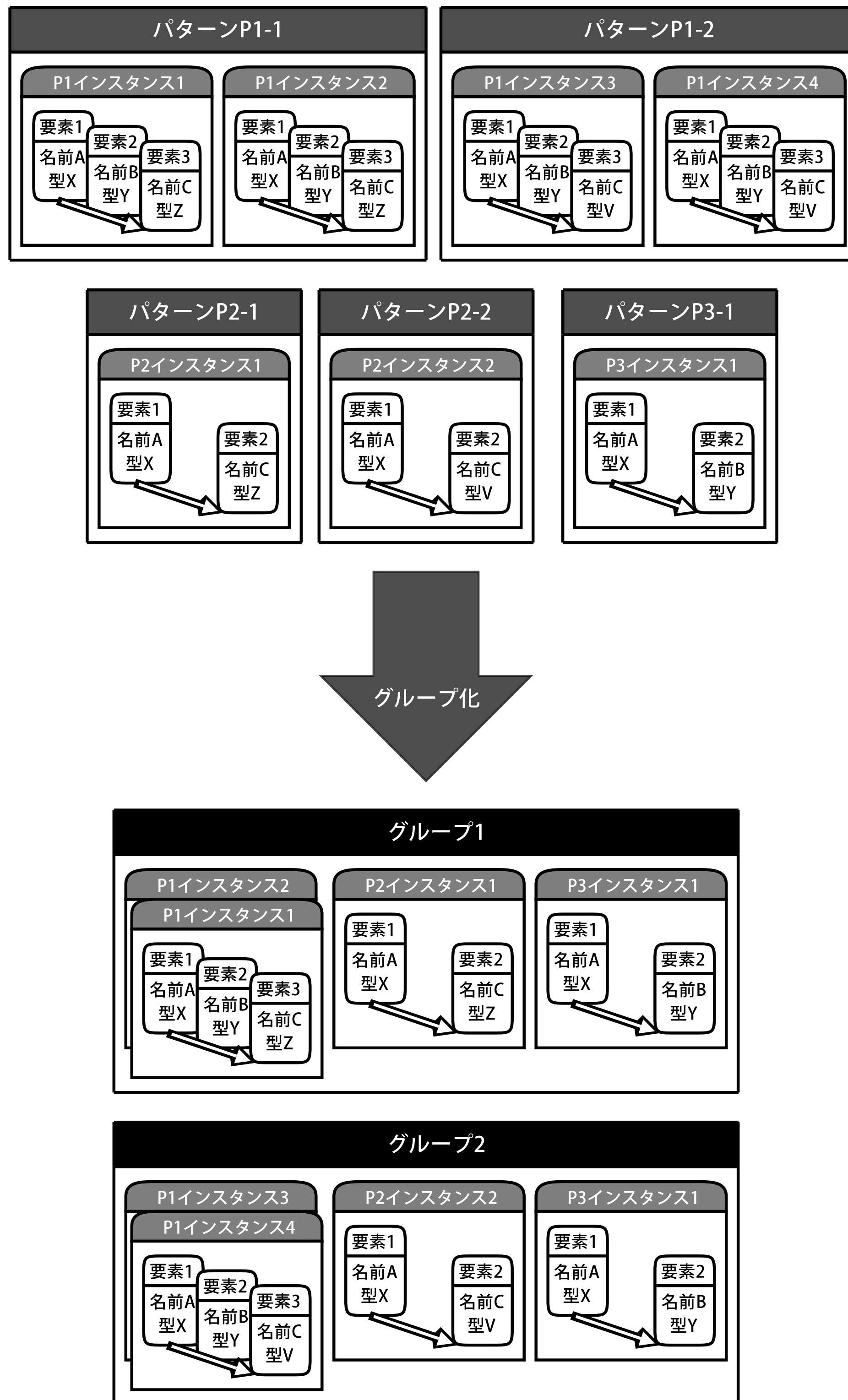


図 12: グループ化

立っていなれば, すなわち確信度が閾値以下であれば, それは規則にしない. しかし, ごく稀に成り立たない場合があれば, 相関ルールが成り立っていないメソッド定義が異常である可能性があり, パターン違反とする.

例えば, 同一グループに含まれるメソッド呼び出しパターン P1, P2 があり, P1 はメソッド定義 {Method1, Method2, Method3, Method4} に, P2 が {Method1, Method2, Method3} に出現したとき, 相関ルール $\{P1\} \Rightarrow \{P2\}$ の確信度は 0.75 となる.

4　適用実験

3 節で述べた手法を実現するツールを実装し, 適用実験を行った. 本節ではその内容および結果と, 結果に対する考察を述べる.

4.1　実験目的

実験目的は以下の 2 つである.

目的 1.　オブジェクト指向プログラムにおいても, 手続き型言語と同様, メソッド呼び出しパターンのパターン違反による欠陥検出が有効か確かめる.

目的 2.　型を考慮することで, 2.6 節で述べたように, 型を考慮しない場合に検出できなかったパターン違反が検出できるようになるか確かめる.

4.2　評価方法

Fung を用いて, JDT バージョン 2 の Core コンポーネント [8](以下 JDT Core と表記する) からメソッド呼び出しパターンを抽出し, 型を考慮する有効性を評価するため, 型を考慮しない場合とする場合との 2 通りでパターン違反の検出を行い, 以下の項目について調査を行った.

1. **目的 1.** を検証するため, 実際に欠陥を含んだパターン違反が検出されるかどうか調べる.
2. **目的 2.** を検証するため, 型を考慮した場合としなかった場合とで検出した欠陥の差分を調査し, 型を考慮することの有効性を調べる.

4.3　実験対象

適用対象の JDT(Java Development Tools) とはオープンソースの統合開発環境である Eclipse[5] に標準で付属する, Java 言語の開発ツールを提供するプラグインである. Core コンポーネントはその中でもソースコードの自動補完や, インクリメンタルコンパイラ, 自動リファクタリングなど様々な機能を実現する. 実験に際し, プログラムの機能に直接関係のないテストコードは除外した. 対象の規模は表 1 に記載した. 行数は, テストコードを除いた数値である.

4.4 実験結果

型を考慮する場合としない場合とでのメソッド呼び出しパターン, グループ, パターン違反の数を表 2 に記す. メソッド呼び出しパターンの最小サポート値は 30, メソッド呼び出しパターンの構成要素の最小値を 4 とし, パターン違反とみなす相関ルールの確信度の最小値を 0.9 とした. また, 検出されたすべてのパターン違反を調査し欠陥の数を調べた.

4.5 考察

前節 4.4 の結果に基づき, 節 4.2 で述べた検証をそれぞれ節 4.5.1, 4.5.2 で行う.

4.5.1 オブジェクト指向プログラムに対するメソッド呼び出しパターンのパターン違反を用いた欠陥検出の有効性の評価

実験により, 型を考慮した場合において 1 つの欠陥が発見された. 以下でその欠陥について説明を行う.

実験により得られた欠陥を図 13 に示す. 出現したファイル名は ConstantPool.java である.

この欠陥は, 次のような相関ルールも含め計 192 のルールに違反している.

$\{writeU1 \rightarrow writeU2 \rightarrow writeU2 \rightarrow writeU1 \rightarrow writeU2 \rightarrow writeU2\}$

$\Rightarrow$

$\{writeU1 \rightarrow writeU2 \rightarrow writeU2 \rightarrow problemReporter \rightarrow referenceType$

$\rightarrow noMoreAvailableSpaceInConstantPool \rightarrow writeU1 \rightarrow writeU2 \rightarrow writeU2\}$

以下, この相関ルールの左辺のメソッド呼び出しパターンを $P1$, 右辺のメソッド呼び出しパターンを $P2$ とよぶ. $P1$ はそのインスタンスが合計 31 回出現しているのに対し, $P2$ は 30 回しか出現していない. さらに, $P1$ が出現する全てのメソッド定義で $P2$ も出現している. パターン違反は, 右辺のメソッド呼び出しパターンが出現するメソッド定義の集合から, 左辺のメソッド呼び出しパターンが出現するメソッド定義の集合を引いた差集合に存在する左辺のパターンのインスタンスである. したがって, この場合は図 13 上段に示した, メソッド定義 `literalIndexForJavaLangObjectGetClass` 中に存在する, $P1$ のインスタンスがパターン違反となる.

行数	ファイル数	メソッド数
334,595	1,654	9,668

表 1: JDT Core の行数, ファイル数, メソッド数

パッケージ: org.eclipse.jdt.internal.compiler.codegen
ファイル　: ConstantPool.java

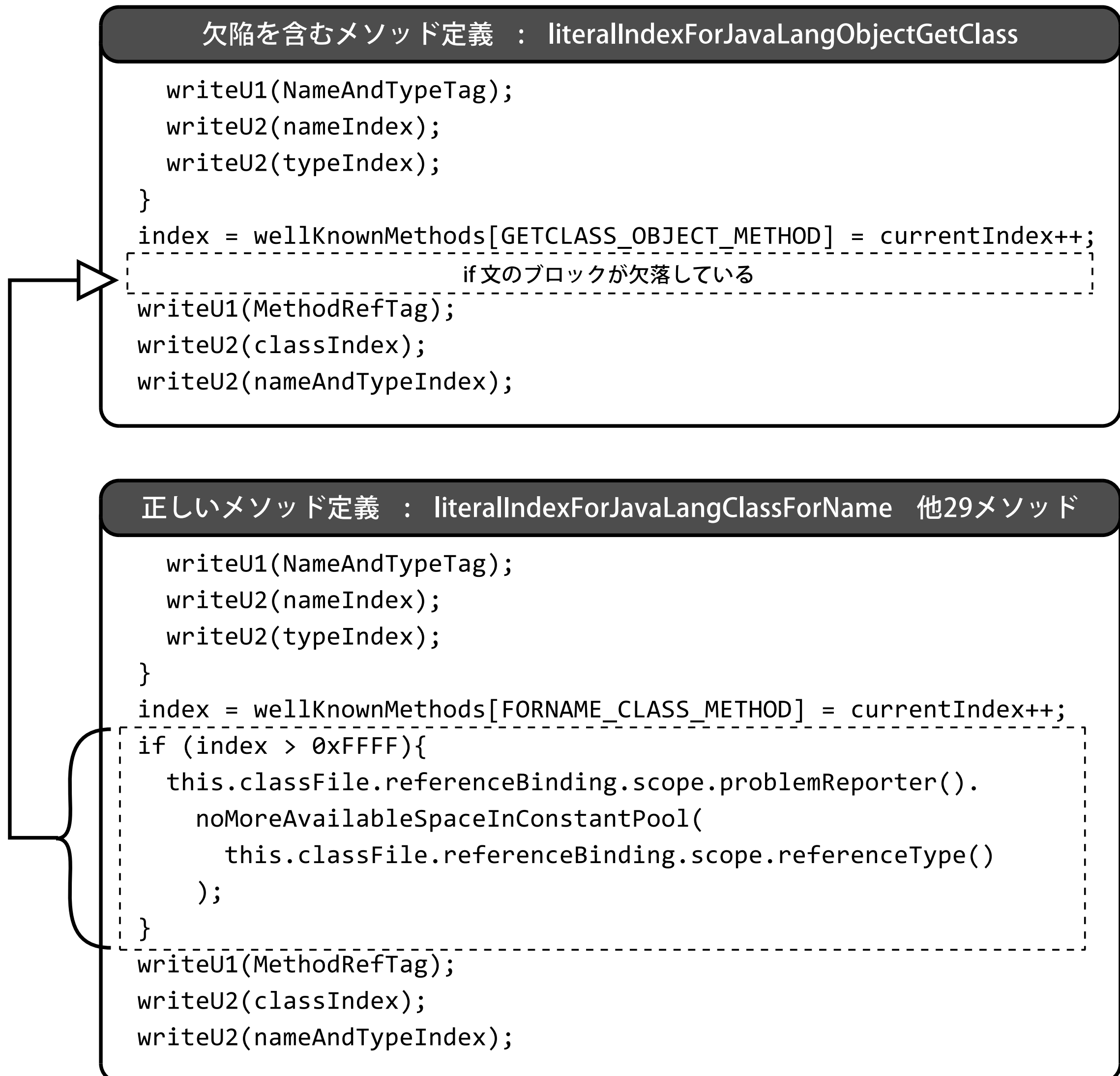


図 13: 欠陥を含むコードと, 正しいコード

図 13 に示したように, メソッド定義 `literalIndexForJavaLangObjectGetClass` 以外の 30 箇所では, `index` に値が代入された後, `index` の値が 16 進数で `FFFF` を上回っていないかチェックし, 上回っていればエラー処理を行っている. しかしメソッド定義 `literalIndexForJava-LangObjectGetClass` ではこれが行われていない. `index` に渡される値を追跡したが, 値が `FFFF` を上回らないための処理などは見当たらず, 欠陥ではないかと判断した.

ただし, JDT のバージョン 3.0.1 で, この欠陥を含むファイルに大規模な改変が加えられ, 欠陥の出現したメソッド定義も含め多数のメソッド定義が削除されていた. 改変に伴うコメントは改変のみに言及しているため, このパターン違反が欠陥であると言う確証は得られなかった.

しかし, 実験で見つかった欠陥のようなコードの欠落による欠陥は起こりうるものである. したがって, オブジェクト指向プログラムにおいても, 手続き型プログラムと同様に, シーケンシャルパターンマイニングにより抽出されたメソッド呼び出しパターンに対して, パターン違反を用いての欠陥検出が可能であることを示すのには十分な例であると考える.

実験では, 型を考慮しない場合で検出できる欠陥が, 型を考慮する場合に検出できなかった例はなかったが, 実際には十分起こり得る. 検出漏れが起こってしまう原因はいくつか考えられる. 例えば, 型を考慮する際にメソッド呼び出しパターンのインスタンスの数が減少してしまい, 最小サポート値に満たない場合などが挙げられる.

4.5.2 型を考慮することの有効性の評価

適用実験では, 表 2 に記載したように, 型を考慮しなかった場合に検出できなかった欠陥が, 型を考慮することで検出できるようになった. これは, 節 2.6.2 で述べた, 相関ルールの左辺のメソッド呼び出しパターンに型の異なるメソッド呼び出しが混入することで出現するメソッド定義が増加し, 相関ルールの確信度が低下してしまうこと原因である.

また, メソッド呼び出しパターン $\{bind \rightarrow bind \rightarrow append \rightarrow bind \rightarrow append\}$ は, 型を考慮しない場合のインスタンス総数が 32 であるが, そのうち 1 つのインスタンスのレシーバ

	型を考慮しない場合	型を考慮する場合
メソッド呼び出しパターン	260	121
グループ	56	13
パターン違反	456	295
欠陥	0	1

表 2: JDT Core から得たメソッド呼び出しパターン, グループ, パターン違反, 欠陥それぞれの数

オブジェクトの型が異なる. そのため, このメソッド呼び出しパターンを左辺に持ち, その他のインスタンスのみを持つメソッド呼び出しパターンを右辺に持つ相関ルールで, 確信度が高くなってしまい, 誤検出の増加につながっている.

いずれの場合も, 型を考慮することで解決された. ゆえに, 型を考慮してパターン違反検出を行うことは, 型を考慮しない場合に対して利点が存在すると言える.

5 まとめ

シーケンシャルパターンマイニングを用いて得られたメソッド呼び出しパターンに対し, メソッド呼び出しパターンのパターン違反から欠陥を検出する手法がある. この手法は手続き型言語にのみ適用例が確認されているが, 本研究ではオブジェクト指向プログラムに適用し, 欠陥を検出できることを示した. オブジェクト指向プログラムへの適用に際し, メソッドを特定できないという問題が考えられたが, メソッド呼び出しパターンを構成するメソッド呼び出しのレシーバオブジェクト, および引数の型を考慮する手法を提案し, 有効性を確認した.

今後の課題としては, 適用したプログラムが現在 JDT Core のみであるため, 他のプログラムにも適用実験を行い, より提案手法の有効性を確実にすることが挙げられる.

また, 評価実験において, 欠陥かどうかの判断をソースコードを追い目視で行った. しかし, 正確を期すためには Bugzilla[3] などの欠陥追跡システムを用いて, 欠陥候補が欠陥として報告されているか調査する必要がある. ただし, 欠陥が報告されていたとしても, 対象となっているバージョンが古いため入手できず, 調査が行えない可能性がある. iBUGS[4] は, Bugzilla を元に構築された欠陥に関する履歴のデータベースで, 欠陥の報告や修正前後のソースコードを閲覧することができる. iBUGS を用いることで, パターン違反により検出した欠陥候補が, 実際に欠陥として報告されているか確認することができる.

本手法では, メソッド呼び出しに関連する型を型階層を無視して比較している. そのため, 共通の親クラスでのメソッドを子クラスがそれぞれ呼び出していて, なおかつ子クラスの両方が親クラスで宣言されていない場合, 同じメソッドを呼び出しているにも関わらず異なるメソッドと判断され, パターン違反などの検出漏れが起こってしまうと考えられる. したがって, 型を考慮する際に型階層を辿りながら比較を行う機能が必要である.

謝辞

本研究において, 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.

本研究において, 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 松下 誠 准教授に深く感謝いたします.

本研究において, 逐次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.

本研究において, 常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 早瀬 康裕 特任助教に深く感謝いたします.

本研究において, 的確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 肥後 芳樹 助教に深く感謝いたします.

本研究において, 数多くの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 吉田 則裕 氏に深く感謝いたします.

本研究において, 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 譜久島 亮 氏に深く感謝いたします.

本研究で使用させて頂いたツール, Fung の製作者である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 伊達 浩典, 三宅 達也 両氏に深く感謝いたします.

最後に, その他様々な御指導, 御助言等を頂い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研究室の皆様に深く感謝いたします.

参考文献

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of VLDB*, Santiago de Chile, Chile, 1994.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of ICDE 1995*, p. 3, Taipei, Taiwan, 1995.

[3] Bugzilla. `http://www.bugzilla.org/`.

[4] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proc. of ASE 2007*, pp. 433–436, 2007.

[5] Eclipse. `http://www.eclipse.org/`.

[6] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proc. of FIMI 2003, Melbourne, FL, USA*, 2003.

[7] D. J. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. The MIT Press, 2001.

[8] JDT. `http://www.eclipse.org/jdt/`.

[9] H. Kagdi, M. L. Collard, and J. I. Maletic. Comparing approaches to mining source code for call-usage patterns. In *Proc. of MSR 2007*, pp. 123–130, Los Alamitos, CA, USA, 2007.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP 1997*, pp. 220–242, Jyväskylä Finland, 1997.

[11] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, Vol. 32, No. 3, pp. 176–192, 2006.

[12] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of ESEC/FSE 2005*, pp. 306–315, Lisbon, Portugal, 2005.

[13] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proc. of ICSE 2000*, pp. 167–176, Limerick, Ireland, 2000.

[14] Fung: A pattern mining tool for java method calls. http://sel.ist.osaka-u.ac.jp/~ishio/fung/.

[15] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto., Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. of ICDE 2001*, pp. 215–224, Heidelberg, Germany, 2001.

[16] パターンワーキンググループ. ソフトウェアパターン入門. ソフト・リサーチ・センター, 2005.

[17] 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, Vol. 2006, No. 35, pp. 49–56, 2006.

[18] 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.