プログラミング言語の構造を考慮した API 利用例検索ツール

A Code Search Tool for API Usage Examples Supporting Program Language Structure

竹之内 啓太 * 石尾 隆 † 井上 克郎 ‡

あらまし ソフトウェア開発において既存のライブラリを利用することは一般的に行われている．しかし，ライブラリの利用は常に容易というわけではなく，開発者の多くは典型的な利用例を説明するコード片を検索，閲覧することで学習を行っている．本研究では，API の利用例を検索するという課題に向けて，(1) トークン列を基本の認識単位としてコメントや空白に影響されず，(2) 構文の区切りに用いられる括弧の対応関係 (変数のスコープ) を考慮し，(3) 識別子の対応関係を開発者が指定可能なソースコードの検索ツールを提案する．検索ツールは非決定性オートマトンを用いたトークン列の認識として実装されており，いわゆる grep コマンドや正規表現検索などと同様に軽量に，データベース構築などの事前準備なしに利用することが可能である．

1 はじめに

ソフトウェア開発において，ライブラリやフレームワークなどの Application Programming Interface (API) の利用が一般的となっている [1]．API を利用することでソースコードの再利用が可能となり，より短い期間で高品質なソフトウェア開発を実現することができる．多くのライブラリには API の仕様が記載されたドキュメントが付属している．たとえば Java 言語用のライブラリのドキュメントには，メソッド単位での入出力や発生する例外の型や値の意味，クラスの階層構造などが記載されており，開発者は API の基本的な利用方法を知ることが可能である．

API の利用は常に簡単というわけではなく，特に開発者にとって不慣れな API の利用には困難が伴うことが知られている [2]．たとえば，API のドキュメントには各メソッドやクラスの仕様が記載されているものの，それらをどのような組み合わせで使用されるべきかまで記載されているわけではない．そのため，API に含まれるあるクラスの複数のメソッドを特定の順番で使用しなければ機能を実現できないような場合は，利用が難しい．また，あるメソッドの引数の型がインターフェースであるが，そのインターフェースに対応するオブジェクトの作成方法が分からない場合なども同様である．開発者が API の利用方法を調査するために多くの時間を費やしてしまうと，結果的にソフトウェア開発の生産性の低下につながる．

開発者の API の利用を支援するために，API の利用例を開発者に提示する様々な手法が研究されている．そのアプローチのひとつが，パターンマイニングを用いた手法である．`UP-Miner` [3] や Saied らの手法 [4] は，ソースコードから同時に使用されやすいメソッドの組み合わせを検出するものである．これらの手法は蓄積された既存ソースコードに対する静的解析であり，API を使用するメソッド呼び出しの列を抽出したのち，それらをクラスタリングすることで使用されるメソッドの組み合わせを求める．得られた結果は開発者が一般的なメソッド呼び出し列を把握する助けとなり，API を用いた実装の生産性の向上に貢献する．同様に，API の使用パターンをマイニングにより求める先行研究は複数存在する [5] [6]．しかしながら，これらの手法はいずれもプログラムの静的解析やクラスタリングなど，計算コスト

*Keita Takenouchi, 大阪大学大学院情報科学研究科

†Takashi Ishio, 大阪大学大学院情報科学研究科

‡Katuro Inoue, 大阪大学大学院情報科学研究科

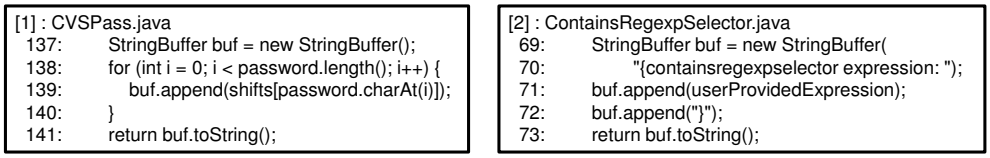

図 1　検索結果の例．それぞれのコード例に対しファイル名と行番号が表示される．

が比較的高い前処理が必要となる．そのため，開発者が特定の API の具体的な使い方を知りたいと感じた際に即座に情報を参照できるようにするには，API の多数の利用例をあらかじめ蓄積し解析しておく必要があり，システムとしての構築が難しい．

パターンマイニングによる抽象化を行わず，具体的な API の利用例を開発者に提示する手法として，ソースコード検索のアプローチが存在する．PARSEWeb [7] は，開発者が注目する特定のクラスのインスタンスの生成方法を調査するためのコード例検索の手法である．検索クエリとして，開発者が現在開発中のプログラムで利用できるクラス $Source$ と実際に利用したいクラス $Destination$ の組 “$Source \rightarrow Destination$” を指定することで，$Source$ 型のインスタンスから $Destination$ 型のインスタンスを生成するようなコード例を検索することが可能である．Strathcona [8] は，開発者が与えたコード片に構造的に類似したコード片を返す手法である．出力となるコード片は入力と同じメソッド呼び出しを持つようなコード片であり，ヒューリスティックなアルゴリズムにより類似度を計算している．これらの手法は，API の使い方のドキュメントには書かれていない部分を補うために有効である一方で,「インスタンスを生成する」や「類似したメソッド呼び出しを持つ」といった限定的な文脈を持つコード例検索の要望に応えるものである．ソフトウェア開発においては，開発者が API を利用する際に遭遇する問題は多岐に渡るため，検索したいソースコードの様々な形に柔軟に対応できる方が望ましい．

本研究では，開発者が様々な API の利用例を迅速に検索するためのソースコード検索ツールを提案する．提案ツールの入力は検索クエリと API を利用しているソースコード集合であり，出力は検索クエリに合致したソースコード片の集合である．開発者が検索したい API の利用文脈はソースコード中のトークンの並びによって指定することを想定する．たとえば，StringBuffer クラスの使い方を調査したいときは，以下のような検索クエリを記述し，最長一致するコード片の検索を指示する．

```
$a = new StringBuffer
??
$a
```

ここで先頭の “$a” は StringBuffer オブジェクトへの参照を代入する変数名の出現を表現しており，続く “=”, “new”, “StringBuffer” はそれぞれ対応するトークンの出現を，2 行目の “??” は 0 個以上の任意のトークンの列を，3 行目の “$a” は先頭に出現した識別子の再度の出現を表現する．この検索クエリに最長マッチするコード片として，ツールは図 1 に示すようなコード片を出力する．この結果から，StringBuffer のインスタンスに対しては append を呼び出してから toString を呼び出す流れを知ることができる．

本検索ツールが持つ重要な特徴の 1 つが，プログラミング言語の持つ構文的な構造，すなわち括弧の対応関係を考慮することである．これにより，出力となるコード片において同名の変数が複数箇所に出現する場合，それらは同じスコープを持つ変数であることが保証される．ツールはクエリの先頭からトークンのマッチを開始すると同時にそのトークンの所属するブロック（開き括弧と閉じ括弧で囲まれた範囲）を認識し，そのブロックの終了までにクエリ全体がマッチしたコード片のみを

最終的な結果として出力する．そのため，同一ファイル内部の他のメソッドで同名の変数が使用されている場合は，それぞれを独立したコード片として認識することになる．これにより，一般的な正規表現検索よりも開発者にとって意味のあるソースコード片を出力する．

検索クエリは，検索対象のプログラミング言語のトークン集合と，3 種類の特殊なトークンの並びによって記述する．基本構成要素は対象プログラミング言語のトークン列であるため，開発者にとって直感的に検索クエリを記述することが可能である．さらに，コード片に対して最長一致あるいは最短一致の選択を許すことにより，開発者の様々な要望に応えるコード例の検索を実現している．検索のアルゴリズムは，検索クエリに対応する非決定性有限オートマトンを構成したのち，検索対象のプログラムのトークン列をオートマトンに入力していくという流れである．この検索のアルゴリズムは一般的な正規表現検索に類似したものであり，検索の前処理はソースコードのトークン列への変換のみであるため，高速に行うことが可能である．

本研究ではケーススタディにより提案ツールの有用性を考察した．その結果，提案ツールは開発者にとって意味のあるコード片を数秒間で検出することが可能であることを確認した．一方で，検索結果となるコード片の中には開発者にとって明らかに有益でない部分が多く含まれる可能性があるなど，今後の課題につながる知見も得られた．

以降，2 章で関連研究について述べ，3 章では提案ツールの仕様について述べる．4 章で検索のアルゴリズムについて，5 章ではケーススタディについて述べる．最後に 6 章でまとめと今後の課題について述べる．

2　関連研究

ソースコードの検索として一般的に知られているのは正規表現 [9] の活用であるが，正規表現は文脈自由文法の構造を表現できないために検索の能力に制限がある．Paul ら [10] によると，まず，`for` や `while` といった予約語が順番に出現することは指定できるが，それらが単純に並んだものと入れ子になっているものを区別できない．変数宣言のようなパターンでも複雑な正規表現になりうるし，パターンがコメント内部のソースコード片にマッチしてしまうこともある．行単位のマッチを行うツールの場合は，途中改行の有無にも影響されてしまう．

検索したいソースコードを表現するためのパターン言語として，Paul ら [10] は SCRUPLE を提案した．このパターン言語は C 言語を想定して宣言，型，変数，関数，式，文に対応するワイルドカードを提供しており，ワイルドカードにマッチした識別子の再度の出現や， for 文や while 文などの構造を指定できる．Matsumura ら [11] は SCRUPLE を用いて，問題を引き起こすようなソースコードの「バグパターン」を記述し，検索するシステムを実現している．構文木に対する専用のパターン言語は強力であるが，利用者は構文木の要素それぞれに対応するワイルドカードを明確に使い分け，マッチするべきプログラム文の制御構造も明示的に指定する必要があることから，学習コストが高いという弱点がある．また，C 言語のプリプロセッサ記述のような文法を無視した要素の影響を受けやすいほか，技術文書に登場する構文的に完全ではないソースコード例などが取り扱えないという制限もある．

Jarzabek [12] は，ソースコードに対する検索用のクエリ言語 PQL を提案した．プログラムおよび手続き，それを構成する変数などの木構造をデータベースに格納し，論理式を満たすようなノード集合やその属性情報をクエリによって抽出することができる．本研究では，検索ツールの利用にあたっての準備を不要にしたいという観点から，データベースを事前に構築するこの方式は採用しなかった．

Deering ら [13] は，解析対象ソフトウェア内部の制御フローや呼び出し関係などの構造をグラフデータベースに変換して検索可能とする解析環境 Atlas を提案した．この環境は応答性を重視しており，開発者がクエリを発行し，その結果を閲覧する

ことでクエリを洗練するという作業を行うことを想定している．本研究のツールも，クエリを素早く記述して実行し，結果を見ながらクエリを洗練できるような応答性を重視した作りとなっている．

Li ら [14] は，あるコード片にバグがあると判明したとき，そのコード片をクエリとして，同じバグが複製されている可能性のある類似コードを検索するツール CBCD を提案した．このツールはプログラム依存グラフとしてコード片の比較を行うことで字句的な表現が異なるコードでも検出することができる．しかし，検索にはあらかじめコード片を用意する必要があるため，API の名前だけが分かっていてその使い方を調査するといった目的では利用できない．

3 提案ツール

本研究では，開発者が不慣れな API を使用する際に必要な様々な調査に利用可能なコード例検索ツールを提案する．提案ツールは，入力として検索対象となるソースファイル集合と検索クエリを受け取り，検索クエリにマッチするコード片の集合を出力する．

検索対象となるソースファイルは，コメントや空白，改行のようなレイアウト情報を取り除いたトークン列として扱う．これは，CCFinder [15] のようなトークン単位のコードクローン検出ツールと同様である．また，プログラミング言語の構文についての情報として，出現順序に対応関係の制約を持つトークン（たとえば “{” と “}”，“(” と “)”）の情報は与えられるものとする．トークン列への変換は字句解析器，対応するトークンの情報は文法定義にのみ依存するため，多くのプログラミング言語に対して適用可能である．本研究では，多くの先行研究にならって対象言語を Java とした実装を行っている．

検索クエリの基本要素は検索対象言語の任意のトークンの列である．コード例を検索したい API の具体的なクラス名やメソッド名が基本となる．検索に柔軟性を持たせるため，以下に示す 3 つの特殊なトークンを導入する．

長さ 0 以上のトークン列「??」 長さ 0 以上の任意のトークン列にマッチする．このトークンを使用することで，あるトークンから他のトークンまでの間のトークン列の差を吸収するような検索クエリを記述することが可能である．

長さ 1 以上のトークン列「 _ 」 長さ 1 以上で，開き括弧と閉じ括弧の対応関係が取れたトークン列にマッチする．複数のマッチ候補がある場合は，最短マッチが採用される．たとえば，“`foo(_)`” という検索クエリは，“(” の後ろに 1 トークン以上続き，そのトークン列で開いた括弧がすべて閉じられた後に出現した “)” でマッチを終了する．そのため，“`foo(a)`” と “`foo(a())`” のようなトークン列にマッチし，文の中におけるトークン列の差を吸収することが可能である．

識別子の対応関係の指定「$変数」 $の後に識別子名が続くようなトークンであり，任意の 1 つの識別子名にマッチする．たとえば，“`$a`” や “`$list`” のようなトークンはそれぞれ$変数であり，それぞれ 1 つの識別子名にマッチする．検索クエリ中に同一の$変数が複数箇所に出現する場合には，すべての出現箇所に対し同じ識別子名がマッチしなければならない．「同じインスタンスに対するメソッド呼び出しの列を知りたい」という一般的な要望にも見られるように，同じ変数 (インスタンス) を同じであると認識することは重要であると考え，本トークンを導入した．

検索クエリが与えられると，4 章に示すアルゴリズムに従って検索クエリに含まれるトークンと検索対象ファイルに含まれるトークン列の比較を行う．可変部分であることが “`_`” や “`$`” を用いて明示的に指定されたもの以外は，トークン単位での文字列の比較が行われる．たとえば，クエリ中で “`a`” という識別子を使ったコード片が指定されると，そのクエリは “`b`” という識別子を使ったコード片にはマッチしない．従って，検索クエリとして上記の特殊トークンを使用せずに検索を行った

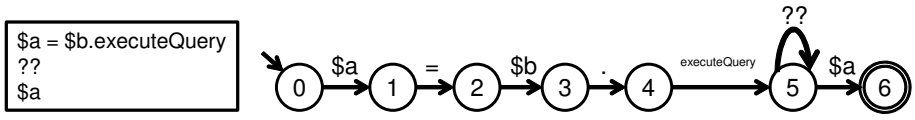

図 2 (左図) 検索クエリ (右図) 対応する非決定性有限オートマトン.

場合，コードクローンの中でもタイプ 1 のクローン [16] 検出と同等の検索となる．

長さ 0 以上のトークン列にマッチする “??” を用いると様々なコード片が検索結果の候補となるが，本研究では同名の変数がすべて同じスコープを持つようなコード片のみを検出する．たとえば，図 1 の [1] に含まれる変数 `buf` はすべての出現において同じスコープを持つため，このコード片は検索結果として適切である．一方で，あるメソッドの中で変数が宣言され，そのメソッドが終了してから別のメソッドで同名の変数が宣言されているようなコード片は，互いにスコープの異なる同名の変数が混在しているため，開発者の意図に合致しないと考える．そこで，検索結果として得られたコード片のうち，対応する開き括弧 (’{’, ’(’, ’[’) が存在しないような閉じ括弧 (’}’, ’)’, ’]’) が出現するようなコード片は，出力から除外する．一方で，対応する閉じ括弧が存在しない開き括弧が存在する場合は，同名の変数が異なるスコープを持つことにはならないため，出力に含まれるものとする．

提案手法の検索において，開発者は検索結果の最短一致と最長一致のいずれかを指定することができる．これにより，開発者の検索したい文脈に応じて出力を選択することができる．たとえば，あるインスタンスに対するすべてのメソッド呼び出しを調査したい場合は，検索は最長一致が適切である．一方で，あるトークン列を 1 つ見つければいいような場合は最短一致が適切である．

4 アルゴリズム

提案手法のアルゴリズムは，クエリに対応する非決定性有限オートマトンの構築，検索対象ファイルのオートマトンへの入力，検索結果の出力の 3 つのステップからなる．

4.1 手順 1．クエリに対応する非決定性有限オートマトンの構築

本手順では，図 2 のように検索クエリから NFA(非決定性有限オートマトン) を構築する．この手順は正規表現検索の一般的なアルゴリズムに類似している．

まずはじめに検索クエリのトークン化を行う．トークン化には構文解析ツール `ANTLR v4`[1] を使用した．本手法の検索クエリは，検索対象言語のトークン集合と特殊な意味を持つ 3 つのトークンの並びで構成される．追加トークンを認識するため，検索対象として選んだ Java の構文規則に 3 つの特殊トークンに対応する規則を追加し，検索クエリに対する字句解析器を生成した．

次に，NFA を構築する．NFA の入力記号は検索対象言語のトークン集合である．NFA の初期状態を生成した後，検索クエリのトークン列を先頭から読み込み，読み込んだトークン t に応じて新たな状態と遷移を追加する．ここで，直前に作成した状態を s_p とすると，トークン t が “??” であるときは，s_p の任意の入力に対する遷移先として状態 s_p 自身を設定する．トークン t が “_” であるときは，新たな状態 s_n を作成し，s_p の任意の入力に対する遷移先として s_n を追加する．また，s_n の任意の入力に対する遷移先として s_n を追加する．トークン t が “??”,“_” 以外であるときは，新たな状態 s_n を作成し，s_p の入力トークン t に対する遷移先として s_n を追加する．トークン列を末尾まで処理した後，最後に作成した状態を受理状態とすれば NFA の構築が完了する．

[1] `http://www.antlr.org/`

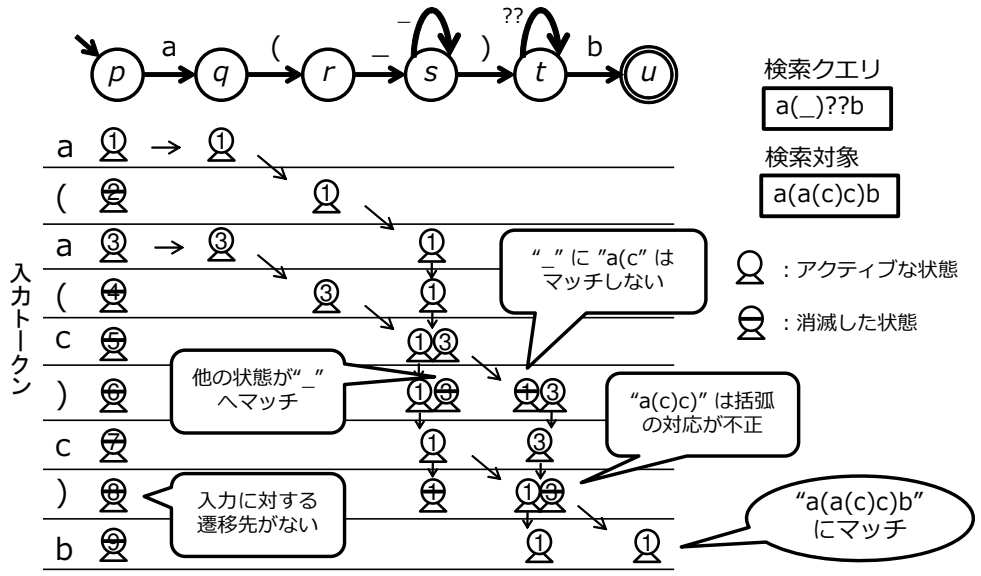

図 3　NFA におけるアクティブな状態の遷移の例．トークンとして a, b, c, (,) を用いている．

4.2　手順 2．検索対象ファイルのオートマトンへの入力

検索はファイルごとに個別に行う．まず，検索対象ファイルを読み込み，検索対象言語の構文規則に従ってトークン化を行う．得られたトークン列に含まれる識別子名が，検索クエリに含まれるすべての識別子名を含んでいる場合のみ，以降の手順を行う．そうでない場合は，そのファイルから検索クエリに一致するコード片が検出されることはないため，次のファイルに進む．

本手法で使用する NFA は一般的な NFA と同様，入力に応じてアクティブな状態が遷移する．アクティブな状態の遷移の例を図 3 に示す．この例では，検索対象の文字列を “a(a(c)c)b” として，検索クエリを “a(_)??b” としたとき，文字列 “a(a(c)c)b” のみが検索にマッチする過程を示している．ただし，「アクティブな状態」は以下の 5 つの情報を保持し，その状況に応じて消滅するルールを持つ．また，同一の情報を保持するアクティブな状態同士は区別しないものとする．すなわち，アクティブな状態の遷移先に，同一の情報を保持するアクティブな状態がすでに存在する場合，それらのアクティブな状態は 1 つのアクティブな状態として扱う．これにより，アクティブな状態数が指数的に増加することを防いでいる．

1. **括弧の対応関係の情報**　括弧の情報を積むスタックであり，入力が開き括弧であるときにプッシュ，閉じ括弧であるときポップされる．このスタックが空の状態でポップが行われる場合，そのアクティブな状態は消滅する．これにより，コード片が独立した複数のブロックにまたがる場合を出力から除外している．受理状態においてこのスタックが空でない場合は，ブロックが閉じられていないというだけで含まれる変数のスコープに問題があるわけではないため，アクティブな状態は消滅しない．なお，検索クエリ自体に，対応する開き括弧を持たないような閉じ括弧が含まれている場合，閉じ括弧に対応する数の開き括弧をあらかじめ初期状態としてスタックに積んでおく．これにより，括弧の対応が取れていないような検索クエリに対する検索を実現している．
2. **“_” にマッチするコード片の括弧の対応関係の情報**　検索クエリ中の 1 つの “_” に対応する 1. と同様のスタックである．アクティブな状態が “_” へのマッチを開始したときに空の状態で作成され，その後の入力トークンの括弧の情報を

積んでいく．“_” へのマッチを終えたとき，すなわちアクティブな状態が次の自己への遷移をもつ状態または受理状態へ到達したとき，スタックが空でなければ，そのアクティブな状態が消滅する．スタックが空であれば同じ “_” へのマッチを続けている他のアクティブな状態を消滅させる．これにより “_” が括弧の対応の取れた最短のトークン列にマッチすることを実現している．

3. **$変数の束縛情報** 検索クエリに含まれる各$変数の束縛情報．ある状態から ”$a” の遷移があり，入力トークンが識別子名 “id” であったとき，”$a=id” という束縛情報を保持する．すでに ”$a” が異なる識別子名によって束縛されている場合，このアクティブな状態は消滅する．これにより検索クエリの$変数にマッチしないものを出力から除外している．
4. **開始行番号** 初期状態から遷移を引き起こした入力トークンの行番号．出力するコード片の開始行に一致する．
5. **終了行番号** 受理状態への遷移を引き起こした入力トークンの行番号．出力するコード片の終了行に一致する．

入力トークンに対する遷移先が存在しない場合は，そのアクティブな状態は消滅する．検索対象ファイルのトークン列を先頭から NFA へ入力していくが，トークンを入力するたびに NFA の初期状態にアクティブな状態を追加する．そして，すべてのトークン列を読み終えるまでに，NFA の受理状態に到達するアクティブな状態の集合を求める．その集合が出力候補となるコード片の集合に対応する．

本手順は提案手法の中でもっとも計算量の大きい手順であり，計算量はアクティブな状態の数に比例する．アクティブな状態の数が最大となるのは NFA 中の全状態が自己への遷移を持つときである．この時のアクティブな状態数を見積もるため，NFA の状態数を M, 入力トークン数を N とする．$変数による束縛の影響を除くと，アクティブな状態は，マッチを開始したトークンの位置（開始行番号），既に入力されたトークン数，その時点での NFA の状態の 3 つ組によって識別される．したがって，$n(1 \leq n \leq N)$ 番目のトークンを読んだ時に生成された（初期状態としてマッチを開始した）アクティブな状態は，その後 $(N-n)$ 回のトークンの入力によって数が増加していくが，その総数はアクティブな状態が NFA の全 M 状態に出現する場合の $M \times (N-n)$ より大きくなることはない．つまり，$O(MN)$ で抑えられる．これがすべての n について同様に言えるため，計算量は $O(MN^2)$ となる．実際は，括弧の対応を考慮しているため N は高々1 ブロック内におけるトークン列の長さとなり，計算量が抑えらている．

4.3 手順 3. 検索結果の出力

本手順では，手順 2. で求めた受理状態へ到達したアクティブな状態から，最短または最長一致するもののみを出力する．基本的には受理状態へ到達したアクティブな状態に対応するコード片を列挙していくだけであるが，これらの中には互いに包含関係にあるコード片が含まれている．検索時に最短一致が指定されている場合は，包含関係にあるコード片のうちもっとも小さいものを，最長一致が指定されている場合はもっとも大きいものを出力する．

まず，アクティブな状態 a_1, a_2 における包含関係 $\subseteq$ を以下のように定義する．ただし，$a.start$ はアクティブな状態 a の開始行番号，$a.end$ は a の終了行番号とする．

$$a_1 \subseteq a_2 \quad \Leftrightarrow \quad a_1.start \geq a_2.start \wedge a_1.end \leq a_2.end$$

この包含関係 $\subseteq$ は半順序関係であるため，アクティブな状態の集合は半順序集合 P となる．最短一致が指定されているときは P の極小元，つまり任意の $x \in P$ に対し $x \subseteq a \rightarrow a = x$ となるような $a \in P$ を求め，ファイルの $a.start$ 行目から $a.end$ 行目までを出力する．同様に，最長一致の場合は P の極大元を出力とする．

(1) : JDBCRealm.java
```
558:    rs = stmt.executeQuery();
559:
560:    if (rs.next()) {
561:        dbCredentials = rs.getString(1);
562:    }
563:    rs.close();
```

(2) : DatabaseInstaller.java
```
894:    ResultSet LxSet = selectLxChildren.executeQuery();
895:    while (LxSet.next()) {
896:        String id = LxSet.getString(1);
897:        String name = LxSet.getString(2);
898:        String parentPath = LxSet.getString(3);
```

図 4　シナリオ 1 の検索結果

5　ケーススタディ

提案手法の有効性について考察するため，2 つのシナリオおいてケーススタディを行った．考察する項目は以下の 2 つである．

- どのような API の利用例が検索可能か
- どのくらいの検索時間がかかるか

検索対象となるクライアントプログラムとして velocity-1.6.4，ant-1.8.4，tomcat-7.0.2 ，webmail-0.7.10，struts-2.2.1，roller-4.0.1 の 6 つのオープンソースソフトウェアのソースコードを用いた．これらに含まれる Java プログラムのファイル数は 5144 個である．また，実行環境として CPU-Intel Xeon，RAM-16GB，HDD-SATA/7200rpm のマシンを使用した．

5.1　[シナリオ 1] インスタンスの操作方法の調査

インスタンスの操作方法は先行研究でよく扱われているトピックである．今回は開発者が Statement インターフェースの executeQuery メソッドの戻り値のインスタンスの操作方法を知りたいという状況を想定する．executeQuery は SQL 文を実行するためのメソッドであり，API ドキュメントから戻り値の型が ResultSet であることは分かる．しかし，ResultSet が持つメソッド数が膨大であるため，どのメソッドによって SQL 文の実行結果を取り出すのかが分からない．この場合，executeQuery の戻り値を扱っているコード例を見たいので以下のような検索クエリを記述する．

```
$rs = _.executeQuery
??
$rs
```

クエリ中の “_” により，“stmt.executeQuery” のような変数に対するメソッド呼び出しだけでなく，“getStmt().executeQuery” のようなメソッドの戻り値に対するメソッド呼び出しも検索することが可能である．また，executeQuery の戻り値に対するすべての操作を知りたいので最長一致で検索する．

その結果，検索は約 2 秒で完了し，図 4 のような検索結果が得られた．全部で 32 個のコード例が検出された．検出されたコード例より，executeQuery メソッドの戻り値のインスタンスに対し next，getString，close などのメソッド呼び出しが行われることが分かる．また，next メソッドは if 文や while 文の条件文として使われることもコード例から読み取れる．パターンマイニングのような抽象化を行う手法では，このような具体的なメソッドの使い方の情報が抜け落ちることがあった．

5.2　[シナリオ 2] インスタンスの生成方法の調査

シナリオ 1 と同様に，インスタンスの生成方法も先行研究でよく取り上げられるトピックである．今回は開発者が Statement インターフェースの実装クラスのインスタンスの生成方法を知りたいという状況を想定する．API ドキュメントには Statement インターフェースの実装クラスの情報が記載されていないため，開発者にとってインスタンスの生成方法が不明である．この場合，Statement インターフェースの executeQuery メソッドのレシーバとなっている変数への代入文を調査

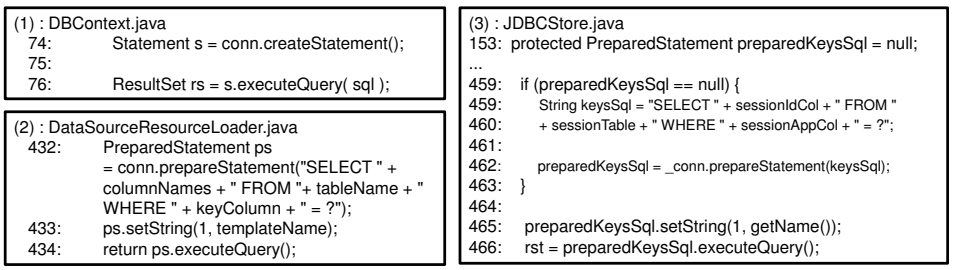

図 5　シナリオ 2 の検索結果

すればよいので，以下のようにクエリを記述する．“`$stmt`” に対する最も近い 1 つの代入文を見たいので，最短一致で検索する．

```
$stmt =
??
$stmt.executeQuery
```

その結果，検索は約 2 秒で完了し，図 5 の検索結果を含む全部で 31 個のコード例が検出された．検出されたコード例 (1),(2) より，`Statement` インターフェースのインスタンスは `createStatement`，`prepareStatement` メソッドにより取得できることが分かる．ただし，今回の検索結果には (3) のような非常に大きなコード片が 2 個含まれた．(3) では検索クエリの “`$stmt =`” がフィールド変数の初期化にマッチし，“`$stmt.executeQuery`” がメソッド内におけるメソッド呼び出しにマッチした．そのため，コード片の多くの部分が開発者にとっては関係のないものとなった．開発者にとって望ましい出力は 462-466 行までのコード片であるが，この部分だけでは括弧の対応が取れていないため，出力からは除外された．

5.3　ケーススタディによる考察

2 つのシナリオによるケーススタディにより，提案手法は開発者にとって有益なコード例が得られることが分かった．また，パターンマイニングなどの手法では抜け落ちる具体的な情報も，提案手法の検索結果から得られることが分かった．一方で検索結果のコード例の中には，開発者の関心と無関係な行が多く含まれる可能性があることも明らかになった．

提案手法は，既存のプログラム群から特定の API の利用例を検索するものである．そのため，検索したいものが限定的なプログラムでのみ使用される API である場合，検索対象となるプログラムの入手が必ずしも容易であるとは限らない．

検索時間は，どちらの場合も前処理込みで 5144 ファイルに対し約 2 秒であった．これは開発者の要望に迅速に応えることができる検索時間であると考えられる．また，検索時間に基づいてインタラクティブな検索をしても開発者のストレスにならない範囲であるといえる．先行研究のような前処理の結果をデータベース等に格納する手法では，提示する情報が事前に計算されている必要があるため，さまざまな要素が組み合わされた要望には対応することができない．一方でインタラクティブな検索では，開発者が検索結果を見ながら検索結果を絞り込むことが可能であり，開発者のより具体的な要望に応えることができる．ただし，提案手法は任意の検索クエリと検索対象プログラムに対し高速に検索が完了するとは限らない．たとえば，検索クエリにマッチするコード片の数が膨大になるような場合は，検索時間が長くなる可能性がある．

6 まとめと今後の課題

本研究では，API 理解を支援するためプログラム群から特定の API を利用しているコード例を検索する手法を提案した．検索はコストの高い前処理なしに高速に行うことができるため，軽量に利用することができる．また，検索クエリとして検索対象言語のトークン集合に特殊なトークンを 3 つだけ追加したものを用いることで，検索クエリの学習の容易さを実現している．ケーススタディでは 2 つのシナリオに基づいた検索を行い，高速に有益なコード例を提示できることを確認した．

今後の課題としては，検索結果から開発者の関心に無関係な部分を除外する手法の改良が挙げられる．また，被験者実験を実施し，検索ツールの有用性やクエリの記述の難易度を評価することも課題として挙げられる．

謝辞 本研究は JSPS 科研費 JP25220003, JP26280021, JP15H02683 の助成を受けたものです．

参考文献

[1] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, pages 280–289, 2013.

[2] Martin P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[3] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *10th IEEE Working Conference on Mining Software Repositories*, pages 319–328, 2013.

[4] Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. Mining multi-level api usage patterns. In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 23–32, 2015.

[5] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 318–343, 2009.

[6] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining api patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 25–34, 2007.

[7] Suresh Thummalapenta. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of ASE*, pages 204–213, 2007.

[8] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of ICSE*, pages 117–125, 2005.

[9] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly Media, 3rd edition, 2006.

[10] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[11] Tomoko Matsumura, Akito Monden, and Ken ichi Matsumoto. A method for detecting faulty code violating implicit coding rules. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 15–21, 2002.

[12] Stan Jarzabek. Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering*, 24(3):197–215, 1998.

[13] Tom Deering, Suresh Kothari, Jeremias Sauceda, and Jon Mathews. Atlas: A new way to explore software, build analysis tools. In *Proceedings of ICSE*, pages 588–591, 2014.

[14] Jingyue Li and Michael D. Ernst. CBCD: Cloned buggy code detector. In *Proceedings of ICSE*, pages 310–320, 2012.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[16] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74:470–495, 2009.