Vol.2010-SE-170 No.9
2010/11/12

コード片に共通した特性を自動抽出するソースコード閲覧ツールの試作

石尾　隆[†1]　井上克郎[†1]

ソフトウェア開発では，開発者がソースコードを検索し，それらを目視で確認する作業が生じる．本研究では，キーワード検索ツールの出力結果を，その該当コード位置のコードが持つ構造あるいは振舞いの属性に応じてグループ化し，共通した属性を持つグループ単位でソースコードを閲覧することができるツールを試作した．

A Source Code Viewer Extracting Common Properties among Source Code Fragments

Takashi Ishio[†1] and Katsuro Inoue[†1]

Software developers inspect source code fragments using a keyword search tool in their daily tasks. In this research, we are developing a source code viewer that classifies source code fragments into groups based on their structural and behavioral properties.

1. はじめに

大規模ソフトウェアのソースコードでは，多数の場所に類似したコード片が出現することがある．この原因として，まず，コード片の複製がプログラミングの最中に頻繁に行われることが知られている[8]．多くは長い識別子を高速に入力するための複製であるが，制御構造や，API の呼び出し順序なども複製して用いられることがある．また，大規模ソフトウェアの開発では，ソフトウェアの設計段階ですべての機能が明らかになるとは限らないため，複数のモジュールに共通した処理があることに気付かず，そのまま個々のモジュールで同じ処理が実装されることがある．また，ロギングや例外処理のような横断的関心事[7] は既存のプログラミング言語ではモジュール化が困難であると指摘されており，結果として類似したコード片がソフトウェアの複数のモジュールに出現することがある．

多数の場所に類似したソースコードが記述されている状況では，該当するコードすべての位置を把握することは困難である．そのため，ソースコードに対するキーワード検索が重要となる．LaToza らは，開発者が実施する日々の作業について調査を実施しており，開発者がソースコードを探索するときの道具の 1 つに `grep` を含めている[9]．

ソースコードの調査において重要な要求の 1 つが，その網羅性である．あるソースコードに誤りが発見されたとき，開発者には，同じように誤りを含んだソースコードを他の場所にも書いているかもしれない，という不安が付きまとう．誤りを含んだコード片と類似したソースコードを検索する手法としては，CCFinder[6] などのコードクローン検出ツールの存在が広く知られているが，これらのツールは，ある閾値以上の長さのソースコード片しか抽出しないという特徴がある．そのため，閾値の設定について十分な経験を積んでいない開発者は，安全に利用することができない．

欠陥の検査などの作業で見逃しを避けたい開発者にとっては，キーワード検索によって発見したソースコードの全検査が，1 つの有力な手段である．キーワード検索では，キーワードの選び方を誤ることで見逃しが発生する可能性はあるが，複数の開発者がキーワードを検討することが可能であり，検査内容の妥当性を検証しやすいという利点がある．その一方で，キーワード検索の弱点は，ソースコードに頻出するキーワードを選択した場合に，多数のソースコード位置が報告されてしまうことである．たとえば，Java で書かれたテキストエディタである JEdit 4.3 において，beep 音を鳴らすコード片を検索するために “beep” をキーワードとして検索すると，全部で 138 行が報告される．また，テキストファイルの保存機能を調べようと “save” を検索すると，設定ファイルの保存機能など，テキストファイルの保存とは無関係な機能を含め，812 行が出力される．これらの検索結果を逐一確認していくことは，開発者にとって大きな負担である．

そこで本研究では，開発者がキーワード検索によって抽出したソースコードを迅速に閲覧，検査するためのツールを試作した．指定されたキーワードを含むソースコードの手続き単位を抽出し，それらの手続きの名前や，手続きの中で呼び出すメソッドなど，様々な観点から共通の属性を抽出し，表形式で提示する．本研究で作成する表は，形式概念分析[14] の用語でコンテキストと呼ばれる形式に従っており，いくつかの手続きに共通する属性や，

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

選択した属性を共通して満たす手続きを，簡単に抽出することが可能となっている．形式概念分析そのものは既存手法であるが，ソースコードの分析に適した属性集合の選び方の提案に本研究での新規性がある．

ツールのアイディアの正しさを確認するために，ケーススタディとして，JEdit における beep 音を鳴らす機能がどのような条件で実行されているかを調査するタスクを実施した．その結果，コード片を分類することが有益であることを確認したが，コード片の分類に用いる属性については，今後も検討が必要である．

以降，2 章では研究背景と関連研究の詳細を解説する．3 章では提案手法について述べる．4 章では実施したケーススタディの結果を記述し，5 章で得られた知見についての議論を行う．6 章で現状のまとめと今後の課題について述べる．

2. 研究の背景

2.1 ソースコードの閲覧手法

キーワード検索は，開発者がソースコードを調査する際の重要な道具である．LaToza らは，開発者が実施する日々の作業について調査を実施しており，開発者がソースコードを探索するときの道具の 1 つに `grep` を含めている[9]．

キーワード検索は，一般に，多数のコード片を出力するため，それを逐次的に閲覧することは開発者にとって大きな負担である．本研究では，`grep` でキーワード検索を行った結果に対し，キーワードの出現位置にあるコード片が持つ共通の属性を明らかにすることで，開発者が迅速に興味あるソースコード群を効果的に閲覧，検査することを可能とする．

キーワード検索の結果を効果的に閲覧する既存手法の 1 つとしては，AspectBrowser がある[13]．AspectBrowser は，ソースファイルをその長さに比例したサイズの縦長の矩形で表現し，プログラム中の全ファイルを 1 画面に並べる．ユーザが指定したキーワードを含む行に対応する位置に着色することで，キーワードが，複数のファイルに渡ってどのように出現しているかを可視化する．単なる矩形による表現ではなく，ソースコードに定義された様々な手続き単位などの情報まで含めてソースコードの概要を表示するツールとしては，Code Map がある[3]．これらの可視化手法は，ソースコードの位置情報に基づいている．検索結果がどのクラス，どのメソッドに所属しているかを知ることは容易であるが，それぞれが実際にどのような処理を行うソースコードであるかは，個別にソースコードを閲覧していく必要がある．本研究では，このような個別のソースコードを無作為に閲覧していくのではなく，構造あるいは振舞いに共通点を持つグループ単位で閲覧することを提案する．

表 1 形式概念分析のコンテキストの例.
Table 1 An example of formal concept analysis.

Object	a1	a2	a3	a4
o1	x	x		x
o2	x	x		x
o3			x	x
o4			x	x

ソースコードの位置に関する共通点を抽出する手法としては，Anbalagan らのポイントカットの自動生成手法[1] が挙げられる．この手法は，Java プログラムにおいて複数のコード片を開発者が指定すると，それらのコード片が特定の名前のメソッドの先頭や末尾に出現しているか，また特定のメソッド呼び出しの前後に出現しているかを調査し，それに対応する AspectJ の execution，call ポイントカットを自動生成する．本研究では，アスペクトへの変換を行うわけではないので，メソッド呼び出しの前後関係については無視して，キーワード検索で発見されたコード片に含まれているメソッド呼び出しの共通点を調査している．取り扱う属性の種類こそ異なるが，コード片に共通した性質を自動抽出するという点で，強く関連した研究である．

ソースコードの振舞いに関する共通点を抽出する手法としては，メソッド呼び出しに関するパターンマイニング手法[4),10] が挙げられる．これらの手法では，1 つのメソッド内で同時に呼び出されることが多いメソッドの集合あるいは列を自動的に抽出し，頻繁に出現するパターンを欠陥検出や横断的関心事の理解に活用する．本研究は，キーワード検索によって選択された手続き群に対する分析である点と，手続き群の位置情報などの属性まで解析対象に加えた点に特徴がある．

ソースコードの類似性に関しては，様々なコードクローン検出手法も提案されている．CCFinder[6] はソースコード上のトークン列，Deckard[5] は構文木を比較して，類似した部分構造を抽出する．これらの手法は，ある閾値以上の長さのコード片を意味のあるコードクローンとして抽出するため，その閾値の値を適切に設定することが難しい．欠陥の検査などの作業で，見逃しを避けたい開発者にとっては，コードクローン検出手法に頼らず，キーワード検索によって発見したソースコードを全検査するというのが，1 つの有力な手段となっている．

2.2 形式概念分析

本研究の特徴は，キーワード検索によって抽出されたソースコード片に対して，それぞ

れのコード片が持つ属性を列挙することである．コード片と属性の対応表は，形式概念分析[14] におけるコンテキストの定義に従っており，いくつかの手続きに共通する属性や，選択した属性を共通して満たす手続きを，対話的な操作により簡単に抽出することが可能となっている．

形式概念分析は，あるオブジェクト集合 O と属性集合 A について，O の各要素が A の各属性のどれを満たしているかをコンテキストという表形式で与えられたとき，そこに含まれるすべての形式概念 (formal concept) を抽出する手法である．オブジェクトと属性のそれぞれ部分集合 O_c, A_c について，O_c の全オブジェクトに共通の属性集合が A_c に等しく，A_c の全属性を満たすオブジェクト集合が O_c に等しいとき，(O_c, A_c) は形式概念であるという．たとえば，**表 1** に示すコンテキストから抽出できる形式概念としては，$c_1 = (\{o1, o2\}, \{a1, a2, a4\})$，$c_2 = (\{o3, o4\}, \{a3, a4\})$，$c_3 = (\{o1, o2, o3, o4\}, \{a4\})$ などが挙げられる．

本研究では，形式概念分析のオブジェクトとして手続きを選び，属性としてソースコードの構造および振舞いに関する情報を使用し，ソースコードの検索結果をコンテキストの形式で可視化している．これにより，開発者は，選択した手続きに共通する属性集合や，選択した属性集合を満たす手続き集合を確実に取得することができる．また，開発者が手続きや属性の部分集合を選ぶことで新たにコンテキストを定義でき，階層的に分析していくことも可能である．

3. 提 案 手 法

本研究では，入力として与えられたソースコードの位置情報に対応する手続き単位の集合 L に対する属性集合 P を定義し，共通の属性を持つ手続き集合の単位でソースコードの閲覧，検査を可能とする手法を提案する．提案手法の入力は，grep の検索結果を想定し，ソースコードのファイル名と行番号の組の集合である．出力は，L に属する各ソースコード位置において P のどの性質が成り立っているかを示す表である．

本研究では対象プログラミング言語として Java を選択しており，属性集合の定義の一部は，Java プログラムの特徴に強く依存する部分がある．

3.1 手続き単位の抽出

grep の検索結果は一般的にはソースコードのファイル名と行番号の組の集合である．本研究では，この結果をまず手続き集合 L に変換し，手続き単位での属性分析を行う．分析単位とするコード断片の候補としては，たとえば前後 N 行を 1 つのコード断片とみなすという方式も検討したが，キーワードが連続した複数行に続けて出現した場合の取り扱いを容易にするために，手続き単位を選択した．

ツールの実装にあたって，本研究では対象プログラミング言語として Java を選択したので，手続き単位とはメソッド，コンストラクタ，クラスのイニシャライザである．本稿では，これら 3 種類の手続き単位をまとめて単に「手続き」と表記する．

Java では，すべての実行可能な命令は，何らかの手続きに所属している．フィールドに対する初期化子をコンストラクタの外に記述することが可能であるが，オブジェクトのフィールド初期化はコンストラクタ `<init>` に，`static` フィールドの初期化はクラスのイニシャライザ `<clinit>` に，それぞれ属するものとして扱う．

grep の検索結果にはコメントも含まれるので，ソースコードのファイル名と行番号の組の集合から，以下のルールに従って，対応する手続き集合 L を求める．

- ソースコードの行番号 l に，実行可能な命令が 1 つ以上存在するとき，それらの命令が所属する手続きを L に含める．
  Java では，改行を行わずにメソッドを定義したり匿名クラスを定義することができるので，1 行に複数の手続きが存在することが可能である．これに対しては，その行番号に命令を持つすべての手続きを L に含めるものとする．
  構文解析を行えば，検索キーワードを持つ構文ノードを発見し，所属手続きを厳密に特定することも可能であるが，実際にそのようなコード記述が登場する可能性は小さいことから，行番号で単純に特定することにした．
- 与えられた行番号 l に対応する実行可能な命令が 1 つもないとき，キーワードはクラスやメソッドの宣言あるいはコメントにマッチしている．このときは，l からそれ以降の行番号を探索し，初めて見つかった実行可能な命令の手続きに所属しているものと考える．これは，手続き冒頭に付与されたコメントが，対応手続きに含まれるという考え方に基づいている．
  この探索は，空白のみの行，複文の終端記号 “}” のいずれかに遭遇した時点で終了する．これにより，メソッドの末尾やクラス宣言に付与されたコメントがキーワードにマッチした場合を取り除く．

上記ルールの適用によって，grep の出力結果は，ソースコードの手続き単位の集合 L へと変換される．どの手続きにも該当しなかった行については，そのまま分析対象からは除外するが，ツールとしては開発者に別途提示するものとする．なお，入力はファイル名と行番号だけなので，grep 以外の検索ツールの出力も，そのまま本手法で分析可能である．

表 2 手続き p の構造情報に関する属性の一覧
Table 2 Structural properties of a procedure p

名前	条件
MethodSig(sig)	手続き p のシグネチャが sig である.
MethodName(n)	手続き p の名前が n である.
MethodPrefix(t)	手続き p の名前の先頭の単語が t である.
MethodSuffix(t)	手続き p の名前の末尾の単語が t である.
MethodKeyword(t)	手続き p の名前の中間に出現する単語に t が含まれる.
Class(c)	手続き p が所属するクラスが c である．ここではパッケージ名を区別する.
SuperType(c)	手続き p が所属するクラスの直接あるいは推移的なスーパータイプが c である.
ClassPrefix(t)	手続き p のクラス名の先頭の単語が t である.
ClassSuffix(t)	手続き p のクラス名の末尾の単語が t である.
ClassKeyword(t)	手続き p のクラス名の中間に出現する単語に t が含まれる.

表 3 手続き p の振舞いに関する属性の一覧
Table 3 Behavioral Properties of a procedure p

名前	条件
Call(m)	手続き p が手続き m を直接呼び出す.
CallName(n)	手続き p が名前 n の手続きを直接呼び出す.
Read(f)	手続き p がフィールド f の値を読みだす.
Write(f)	手続き p がフィールド f に値を書き込む.
Access(f)	手続き p がフィールド f の値を読むか書くかする.

3.2 属性集合

手続き単位の集合 L に対して，共通する属性を取り出す．本研究では，属性集合 P として，以下の 2 種類を用いる．

- ソースコードの構造情報に関する属性．たとえば，2 つの手続きが同じクラス c に宣言されているのであれば，それら 2 つの手続きは「クラス c に宣言されている」という共通した属性を持つと考える．
- ソースコードの振舞いに関する属性．たとえば，2 つの手続きが同じメソッド m を呼び出していれば，それら 2 つの手続きは「メソッド m を直接呼び出している」という共通した属性を持つと考える．

ソースコードの位置に関する属性を**表 2** に，振る舞いに関する属性を**表 3** にそれぞれ示す．各手続き $p \in L$ に対して成り立つ属性の計算は，p が所属するクラスやそのスーパータイプの情報，p が呼び出すメソッド集合などを求めることにより得られる．表 2，表 3 で使用している表現は，以下の通りである．

- コンストラクタの**名前**は，その所属クラスに関わらず `<init>` とする．
- クラスのイニシャライザの**名前**は，その所属クラスに関わらず `<clinit>` とする．
- 手続きの**シグネチャ**は，名前，引数の型，戻り値の型の並びである．手続きを宣言するクラス名は含まない．シグネチャではなく「手続き」の比較を行う Call では，メソッドの宣言クラスまで含めて区別している．
- 手続きが所属する**クラス**は，パッケージ名を含む完全修飾名で区別する．
- クラスの**スーパータイプ**はクラス，インタフェースの両方を含むものとする．
- フィールドは，その所属クラス，型名，変数名によって識別する．
- 手続きおよびクラスの名前を構成する**単語**は，名前を `CamelCase` 型であると仮定して単語列に分解したとき，すなわち小文字の次に出現する大文字を単語の開始位置だとみなす方式により単語に分解することで計算する（たとえば `methodCamelCase` であれば `method`，`Camel`，`Case` の 3 単語と分解される）．中間に出現する単語とは，先頭でも末尾でもない単語を指す．1 単語しかない名前については，その単語が先頭の単語であり，同時に末尾の単語となる．

使用している属性集合は，解析対象の Java プログラムと手続き集合 L に対して，すべての属性をあらかじめ列挙することなしに，成立する属性だけを効率的に計算することが可能なものを選定している．まず，手続き p に対して，そのシグネチャ sig から，MethodSig(sig) が成り立つことが分かる．そのメソッド名から，MethodName を計算することができ，また，メソッド名を単語列に分割することで，メソッド名に対して成立する MethodPrefix, MethodSuffix, MethodKeyword の 3 種類の属性を計算できる．同様に，手続き p はファイル名と行番号によって指定されているので，所属するクラス c を一意に特定し，Class(c) を得る．また，クラス名を単語列に分解することで，ClassPrefix, ClassSuffix, ClassKeyword の属性を計算する．そして，クラス c からスーパータイプの情報を取得し，`java.lang.Object` までのすべての継承関係を調査すれば，SuperType 属性が計算できる．

振舞いに関する属性は，呼び出し関係と，フィールド参照から計算される．動的束縛の解決には，CHA[2] を適用し，可能性のある呼び出しすべてを列挙する．CHA は，RTA や VTA などの手法では取り除くことができる「実際には起きえない」呼び出し関係を抽出する可能性がある[12] が，クラス階層の情報さえあれば計算可能であり，L に含まれないコードの解析を必要としない点で効率的である．また，リフレクションが使われているプログラムに対しても適用可能である．

属性の計算に使用した情報を整理すると，次の通りである．

表 4 属性間の含意関係
Table 4 Implications among Properties

MethodSig	→	MethodName
MethodName	→	MethodPrefix
MethodName	→	MethodSuffix
MethodName	→	MethodKeyword
Class	→	SuperType
Class	→	ClassPrefix
Class	→	ClassSuffix
Class	→	ClassKeyword
CallSig	→	CallName
Read	→	Access
Write	→	Access

- 対象 Java プログラムにおけるクラス定義の一覧.
- それらのクラス間での継承関係.
- 各クラスでのメソッド宣言の一覧.
- 手続き集合 L の処理に出現するメソッド呼び出しの一覧と，フィールド読み書きの一覧.

これらの情報の抽出には，Java バイトコード解析を適用している．また，CHA を用いたメソッド呼び出しの解決は，`equals` や `hashCode` など，広範囲に渡ってオーバーライドされるメソッドに対して膨大なメソッドの一覧を出力してしまう可能性があるので，これらのメソッドについては結果に含めないよう，経験的なフィルタを適用可能にしている．

属性の抽出が完了したら，最後に，分析を行う開発者にとって意味のない属性を取り除く．まず，L に属する手続きのうち，高々1 つの手続きだけが満たすような属性は，抽出するコード片をグループ化するという目的に対して不要であるため，属性集合から取り除く．続いて，属性を満たす手続きの集合が完全に一致するような属性の組 $p1$ と $p2$ が存在するとき，それらを 1 つの属性 $p1 \wedge p2$ に置換する．ただし，**表 4** に示すように，いくつかの属性の組については，一方の属性を持つ手続きは必ず他方の属性を持つことがある．このときは，最も制約が強い条件（含意関係における十分条件）を残すことにし，$p1 \rightarrow p2$ であるときは，$p1 \wedge p2$ は単に $p1$ とし，$p2$ を P から取り除く．たとえば，標準出力を意味する `System.out` フィールドに対して代入が行われることがなく，Read(`System.out`) と Access(`System.out`) を満たす手続き集合が一致したとする．このとき，Read(`System.out`) だけを P の要素として残す．

表 5 JEdit に対してキーワード beep を検索したときのコンテキストの一部.
Table 5 A snippet of a context for JEdit with a keyword “beep”.

Procedure	c1	c2	c3	c4
Abbrevs.expandAbbrev	x	x		
EditPane.goToNextMarker	x		x	
EditPane.goToPrevMarker	x		x	
JEditBuffer.undo	x	x		x

where
c1 = CallName(beep),
c2 = CallName(isEditable),
c3 = CallName(moveCaretPosition),
c4 = Access(UndoManager JEditBuffer.undoMgr).

3.3 コンテキストの可視化

手続き集合 L と，属性集合 P が決まり，L に属する各手続きが満たす属性の一覧が得られたので，これをコンテキストとして可視化する．JEdit に対してキーワード “beep” を検索したときに得られるコンテキストの抜粋を**表 5** に示す．コンテキスト表の一般的な方式に従い，1 行を 1 つの手続きに，1 列を 1 つの属性に割り当てており，ある手続きがある属性を満たすことを記号 “x” で表現している．また，スペースの都合上，メソッドの表記については，所属クラスのパッケージ名や引数を省略している．

本研究では多数の手続きと属性を扱うことから，手続き，属性をそれぞれ適切な順序で並べる必要がある．手続きの表示順序には，パッケージ名，クラス名を連結したときのアルファベット順を，メソッド名はキーワード検索で発見されたときの行番号順を採用した．Java ではパッケージ名とクラス名がそのままディレクトリ名とファイル名に対応することから，`grep` による検索結果をアルファベット順でソートして出力した順序に合致する．属性の順序については，多数の手続きに共通する属性ほど重要な属性であると考え，表として閲覧しやすい左側に配置することにした．多数の手続きに出現するほど重要である，というのは Marin らによる Fan-In 解析におけるランキング手法である[11]．Marin らの発想は，多数のメソッドから呼ばれているメソッドほど横断的関心事である可能性が高い，というものであるが，本研究でもメソッド呼び出し関係を扱っており，多数の手続きに共通するメソッド呼出しなどの属性は，より重要であると考えた．

3.4 コンテキストの分析

コンテキストを用いて対話的にソースコード検査を行うツールの実際のスクリーンショットを**図 1** に示す．3.3 節で述べた方式でコンテキスト表を可視化しており，ユーザは，この

Vol.2010-SE-170 No.9
2010/11/12

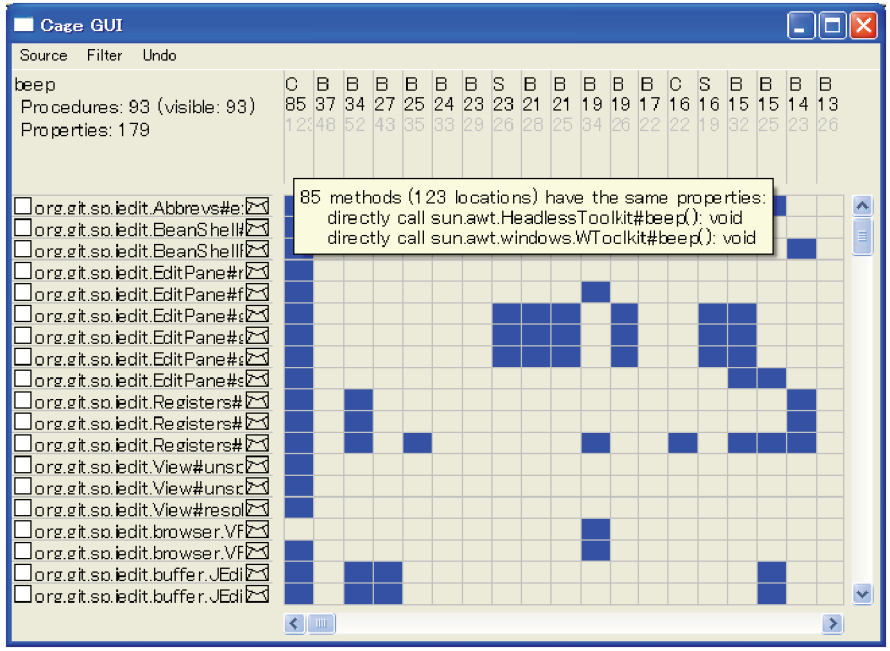

図 1 ツールが可視化したコンテキストのスクリーンショット.
Fig. 1 A screenshot of our tool that shows a formal context.

コンテキスト表から以下の操作を行うことができる.

属性による手続きの並べ替えとフィルタリング. 列をクリックすることで，指定の属性を持つ手続きだけ表示することができる．フィルタリングした結果に対して，現在表示中の手続き群の範囲で多くの手続きが満たす属性を優先し，属性を並べ替えることもできる.

選択したソースコードに共通する属性の自動抽出. 選択したすべての手続きが共通して持つ属性の値を取り出し，その属性によってフィルタリングを行う．たとえば表 5 において `EditPane.goToNextMarker` を選択したとすると，$\{c1, c3\}$ がフィルタリングの条件となり，条件を満たす `EditPane.goToNextMarker` と `EditPane.goToPrevMarker` が表示された状態になる．この操作は，形式概念分析において，選択した手続き群を含む形式概念を求めることに等しい.

手続きのソースコードの表示. 選択した任意の手続きのソースコードを表示する．検索キーワードにマッチした文字列を強調表示することで該当位置を素早く認識できる．また，ソースコードを表示したかどうかはツール側で記録しておき，未閲覧のソースコード集合に共通する属性が何であるかを調査可能としている.

4. ケーススタディ

作成したツールを用いて，JEdit のコードを分析した事例について述べる．JEdit は Java で書かれたテキストエディタであり，ユーザが操作に対して beep 音を鳴らすことがある．本ケーススタディでは，JEdit 4.3 のソースコードを分析し，音が鳴る要因を調査した.

検索キーワードに “beep” を選んで `grep` を実行すると，138 行のコードを発見することができる．これらの行は，93 個の手続きに該当し，2 つ以上の手続きに共通した属性は 181 個であった．処理時間は，Intel Core2 Duo U9300 1.20GHz のノート PC 上で Java 仮想マシンに 1GB のメモリを割り当てた場合で，約 6 分であった．抽出された属性のうち，それを満たす手続きの数が上位 5 件の属性を調べると，次のようになる（スペースの都合上，JEdit のパッケージ名は省略している）.

```
87 methods: Call(void sun.awt.windows.WToolkit.beep()) &&
            Call(void sun.awt.HeadlessToolkit.beep())
37 methods: Read(JEditBuffer TextArea.buffer)
34 methods: CallName(isEditable)
27 methods: Call(boolean JEditBuffer.isEditable())
25 methods: CallName(getLineStartOffset)
```

このうち，第 1 位の属性から，93 個の手続きのうち 87 個が，実際に `beep` メソッドを呼び出していることが分かる．残る 6 個を確認したところ，いずれもコメントでの出現であった．第 2 位のフィールド参照は，JEdit が持つテキストを管理，操作するバッファオブジェクトを読み出す操作であり，これ自体は強い手がかりとはならなかった.

第 3 位と第 4 位は，`isEditable` という同じメソッドが原因になって抽出されており，複数のクラスに `isEditable` というメソッドが定義されていることが推測できる．CallName(isEditable) を満たす手続き群だけを表示させると，次のような手続き群が得られる.

```
boolean Abbrevs.expandAbbrev(View, boolean)
void Registers.cut(TextArea, char)
void Registers.paste(TextArea, char, boolean)
```

```
boolean SearchAndReplace.replace(View)
void JEditBuffer.undo(TextArea)
void JEditBuffer.redo(TextArea)
void CompleteWord.completeWord(View)
void TextArea.backspaceWord(boolean)
void TextArea.formatParagraph()
void TextArea.joinLines()
void TextArea.lineComment()
void TextArea.userInput(char)
...
```

これらのメソッドは，テキスト編集操作を行うメソッドである．JEditBuffer クラスの undo メソッドの先頭部分を抜粋したものを以下に示す．

```
public void undo(...) {
    if(undoMgr == null)
        return;

    if(!isEditable()) {
        textArea.getToolkit().beep();
        return;
    }
    ... // undo the previous action
}
```

このように手続き群を確認したところ，いずれもテキストバッファが編集可能かどうかを検査しており，編集不可能なバッファに対して操作を行おうとした場合に beep 音を鳴らし，編集操作を行わないようにしていることが確認できた．

CallName(isEditable) について調査を行ったので，CallName(isEditable) を満たさない手続きのみに探索範囲を絞った表を取り出し，多数の手続きに共通する属性を確認したところ，以下の属性が上位に出現した．

```
53 methods: Call(void sun.awt.windows.WToolkit.beep()) &&
            Call(void sun.awt.HeadlessToolkit.beep())
19 methods: MethodKeyword(To)
18 methods: CallName(moveCaretPosition)
```

`isEditable` を呼び出すメソッドを除外したので，残り 53 メソッドを調査すれば良いことが分かる．2 番目の属性は，`goToPrevMarker` や `narrowToSelection` のようにメソッド名の共通キーワードを示している．3 番目の属性は，`moveCaretPosition` というカーソル移動のためのメソッド呼び出しを示していた．調査してみたところ，MethodKeyword(To) と CallName(moveCaretPosition) はほぼ同じメソッド集合を指しており，goToNextBracket, goToNextCharacter，goToPrevWord のように，前後の特定の位置までカーソルを移動するメソッドを含んでいた．これらのメソッドでは，たとえばテキスト領域にマーカーが置かれていないのに「次のマーカーへ移動する」という指示を与えられてしまいカーソルを移動できない，という事態が生じた場合に beep 音を鳴らしていることが判明した．

ここまで，全部で 7 つの属性について調べた段階で，既に 87 の `beep` メソッドの呼び出しのうち，過半数の 52 個の調査が完了したことになる．残っている手続き群は，Call(TextArea View.getTextArea()) など，きわめて一般的なメソッドを呼び出すのみで，共通の属性はほとんど見つからなかった．そのため，ここからはクラス単位で，たとえば `HistroyText` クラスに書かれた検索機能メソッド 4 つを順番に調べるなど，個別に実施する必要があった．これらのメソッドには，全体として「ユーザに指示された操作が実行不可能なとき」beep 音を鳴らすという共通点はあったが，キーワードがないのに検索しようとした，ヒストリがないのにそれを操作しようとしたなど，それぞれ判定条件が大きく異なっていたため，グループ単位でまとめて調査することができなかった．

5. 議　　論

本研究では，属性として，メソッドの名前，クラスの名前，クラス階層，メソッドの呼び出し関係，フィールドの参照関係を使用したが，JEdit に対する分析では，属性のうち Call, CallName の 2 つが大きな役割を担った．キーワードで発見された個々の手続きはそれぞれ異なる処理を実行しており，たとえば `isEditable` を呼び出すなどといった小さな共通点から，それらの一群の処理の特徴を理解することが多かった．このような処理を特徴づけるメソッドがどの程度存在するのか，今後も調査が必要である．

著者らは他の 10 万行程度のソフトウェアに対してもコード読解作業に本ツールを適用しているが，その作業でもやはり Call, CallName の 2 つは有用であり，次いで SuperType や ClassSuffix によるコード位置の分類が有効に働いた事例があった．これらの情報は通常の `grep` の結果からは即座に読み取ることができないので，共通の属性として提示すること

が有益であったと考えている．

本研究で使用した属性群は，AspectJ におけるポイントカットの概念と強いつながりがある．たとえば，本研究での Call 属性にあたるメソッド呼び出しの存在は， AspectJ では call ポイントカットとして表現されているし，あるメソッドの中のコードであることを示す属性である MethodSig には，execution ポイントカットが対応していると考えられる．AspectJ において，横断的関心事の表現に call などのポイントカットが多く使われていることが，本研究において，手続き集合の特徴の理解を Call などに頼ったことと関係がある可能性がある．

6. ま と め

本研究では，キーワード検索によって抽出された手続き集合に対して，共通する属性を抽出し，表形式で可視化，分析するツールを試作した．今後，推移的なメソッド呼び出しなど，多様な属性の導入について検討を行うとともに，ソフトウェアの理解に役立つ属性が何であるのか，調査を継続していく予定である．

謝辞 本研究は，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) Anbalagan, P. and Xie, T.: Automated Inference of Pointcuts in Aspect-Oriented Refactoring, *Proceedings of the 29th International Conference on Software Engineering*, pp.127–136 (2007).
2) Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis, *Proceedings of the 9th European Conference on Object-Oriented Programming*.
3) Deline, R., Venolia, G. and Rowan, K.: Software Development with Code Maps, *Communications of the ACM*, Vol.53, No.8, pp.48–54 (2010).
4) Ishio, T., Date, H., Miyake, T. and Inoue, K.: Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs, *Proceedings of the 15th IEEE Working Conference on Reverse Engineering*, pp.123–132 (2008).
5) Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones, *Proceedings of the 29th International Conference on Software Engineering*, pp.96–105 (2007).
6) 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).
7) 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).
8) Kim, M., Bergman, L., Lau, T. and Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL, *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pp.83–92 (2004).
9) LaToza, T.D. and Myers, B.A.: Developers Ask Reachability Questions, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp.185–194 (2010).
10) Li, Z. and Zhou, Y.: PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code, *Proceedings of the 13th Symposium on Foundations of Software Engineering*, pp.306–315 (2005).
11) 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).
12) Qian, F. and Hendren, L.: A study of type analysis for speculative method inlining in a JIT environment, *Proceedings of the 14th International Conference on Compiler Construction*, Springer, pp.255–270 (2005).
13) Shonle, M., Neddenriep, J. and Griswold, W.: AspectBrowser for Eclipse: a case study in plug-in retargeting, *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, New York, NY, USA, ACM, pp.78–82 (2004).
14) 鈴木 治，室伏俊明：形式概念分析: 入門・支援ソフト・応用，日本知能情報ファジィ学会誌， Vol.19, No.2, pp.103–142 (2007-04-15).