特別研究報告

題目

既存ソフトウェア中の頻出コード片を用いたコード補完手法の提案

指導教員
井上 克郎 教授

報告者
関山 太朗

平成 22 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 21 年度 特別研究報告

既存ソフトウェア中の頻出コード片を用いたコード補完手法の提案

関山 太朗

内容梗概

多くの開発者がコード補完を利用してソフトウェアの開発を行なっている．コード補完はキーワードからコード片を生成する機能で，開発者はコード補完によりソースコードに出現するパターンの雛型を取得することができる．しかし，既存のコード補完システムでは，ソフトウェアそれぞれに存在するパターンから雛型を生成することはできない．そのため，ソフトウェアに再利用可能なパターンが存在する場合でも，開発者がそのことを知らなければ，開発者は他のソースコードと類似した処理を再度記述することになる．しかし，既存の処理と類似した処理を再度記述することは欠陥の混入や開発コストの増大につながる．

そこで本研究では，ソフトウェアに存在するパターン情報を基に，開発者が編集しているソースコードの補完を行うコード補完手法を提案する．提案手法では，まず開発者が入力した目的の機能を表すキーワードに関連したパターン情報を検索する．そして，検索したパターンを基にソースコードの編集状況に適合したコード片を生成し，それを含むコード例を開発者へ自動的に推薦する．

本研究では，プログラミング言語 Java で開発されたソフトウェアを対象として，提案手法を組み込んだ開発環境を試作した．評価実験では，キーワードを用いたパターンの検索方法の妥当性を検証するために，4 つのオープンソースソフトウェアから得た 180 件から 7500 件ほどの各パターン集合を対象として，目的のパターンを取得できることを確認した．また本手法で生成したコード片が妥当であることを検証するために，生成したコード片と，そのコード片を生成するときに用いたパターンが出現するソースコードを比較した．その結果，特定のケースを除いて両者の間に大きな違いがなかったことを確認した．

主な用語

コーディングパターン
コード生成
ソースコード検索

目 次

```
for ([iterable type] [iterable element] : [iterable]) {
}
```

図 1: Eclipse のコード補完機能の使用例

```
public class Abbrevs {
  ...
  public staic boolean expandAbbrev(...) {
    Buffer buffer = view.getBuffer();
    ...
    if (!buffer.isEditable()) {
      view.getToolkit().beep();
      return false;
    } ... } ...
}
```

```
public class CompleteWord extends ... {
  ...
  public static void completeWord(View view) {
    ...
    JEditTextArea textArea = ...
    Buffer buffer = ...
    if (!buffer.isEditable()) {
      textArea.getToolkit().beep();
      return ;
    } ... } ...
}
```

```
public class JEditBuffer {
  ...
  public void undo(JEditTextArea textArea) {
    ...
    if (!isEditable()) {
      textArea.getToolkit().beep();
      return ;
    } ... } ...
}
```

```
public class TextArea extends JComponent {
  protected JEditBuffer buffer;
  ...
  public void deleteLine() {
    if (!buffer.isEditable()) {
      getToolkit().beep();
      return ;
    } ... } ...
}
```

図 2: jEdit4.3pre11 に存在するパターンの例

1 まえがき

多くの開発者がコード補完を利用してソフトウェアの開発を行っている．コード補完はキーワードからコード片を生成する機能で，開発者はコード補完によりソースコードに出現するパターンの雛型を取得することができる．例えば Eclipse[2] のコード補完では，“foreach” というキーワードから，図 1 のようなコレクションの各要素を順に取得するというパターンの雛型を取得することができる．この雛型をコード片として利用するには，iterable type にコレクションの要素の型を，iterable element にコレクションの要素に対応する変数名を，iterable に参照する要素をもつコレクションを指定し，指定したコレクションの各要素に対して行う処理をブロックの中に記述する．

しかし，既存のコード補完システムでは，ソフトウェアそれぞれに存在するパターンから雛型を生成することはできない．ソフトウェアに存在するパターンの例を図 2 に示す．図 2 の 4 つのソースコードはテキストエディタ jEdit4.3pre11[4] から得られたもので，それぞれテキストに対して変更を加える処理を行うメソッドが記述されている．図 2 に示した 4 つのソースコードには，いずれも次の要素が順序通りに出現している．

1. JEditBuffer オブジェクトに対するメソッド boolean isEditable() の呼び出し
2. if 文の開始
3. JComponenet オブジェクトに対するメソッド Toolkit getToolkit() の呼び出し
4. Tookit オブジェクトに対するメソッド void beep() の呼び出し
5. if 文の終了

この例から，jEdit4.3pre11 にはテキストの変更ができないときは音を鳴らしてユーザーに通知するという処理がパターンとしてソースコード上に出現していると推測できる．

既存のコード補完システムでは，このようなパターンの雛型を取得することができないため，ソフトウェアに再利用可能なパターンが存在する場合でも開発者がそのことを知らなければ，開発者は他のソースコードと類似した処理を再度記述することになる．例えば jEdit4.3pre11 に新しいテキスト編集処理を追加するとき，開発者はテキストが編集できなければ音を鳴らすという処理を記述する必要がある．しかし，そのような処理を図 2 のように実装していることを開発者が知らなければ，開発者は新しく同じ機能をもつ処理を記述しなければならない．しかし，既存の処理と類似した処理を再度記述することは欠陥の混入や開発コストの増大につながる．

そこで本研究では，ソフトウェアに存在するパターン情報を基に，開発者が編集しているソースコードの補完を行うコード補完手法を提案する．提案手法では，まず開発者が入力した目的の機能を表すキーワードに関連したパターン情報を検索する．そして，検索したパターンを基にソースコードの編集状況に適合したコード片を生成し，それを含むコード例を開発者へ自動的に推薦する．

我々の研究グループでは，既存のソースコード群から頻出するコード片を表現するコーディングパターンを抽出する手法を提案している [9, 19]．コーディングパターンは既存ソースコード群で頻出するコード片を表現したものであるため，ソフトウェアから抽出したコーディングパターンの中には，ソフトウェア独自のパターンが含まれていると考えられる．そのため，本研究ではソフトウェアから抽出したコーディングパターンをソフトウェア独自のパターンとして扱う．図 3 は図 2 の 4 つのソースコード片から得られるコーディングパターンの例である．JEditBuffer.isEditable() は JEditBuffer オブジェトに対するメソッド isEditable() の呼び出しに対応し，IF，END-IF はそれぞれ if 文の開始と終了に対応している．このように，コーディングパターンは複数のソースコードに共通して出現するメソッド呼び出しと IF などの制御構造要素の列を抽出したものである．以降ではソフトウェア独自のパターンをソフトウェアのパターンとよび，単にパターンとだけ記述するときはコーディングパターンを表す．

```
JEditBuffer.isEditable()
IF
JComponent.getToolkit()
Toolkit.beep()
END-IF
```

図 3: 図 2 のソースコードから抽出したコーディングパターン

```
public void editBuffer()
{
  org.gjt.sp.jedit.buffer.JEditBuffer buf = this.getBuffer();

  buffer editable
}
```

(a) 適用前

```
public void editBuffer()
{
  org.gjt.sp.jedit.buffer.JEditBuffer buf = this.getBuffer();
  boolean var0;
  var0 = buf.isEditable();
  if (var0) {
    javax.swing.JComponent var1;
    java.swt.Toolkit var2;
    var2 = var1.getToolkit();
    var2.beep();
  }
}
```

(b) 適用後

図 4: プログラミング言語 Java への本手法の適用例

本研究では，プログラミング言語 Java で開発されたソフトウェアを対象として，提案手法を組み込んだ開発環境を試作した．このツールの使用例を図 4 に示す．図 4(a) は開発者が新しく記述しようとしているテキスト編集処理の断片である．メソッドの 3 行目にある “buffer editable” が本手法での開発者が入力するキーワードであり，図 4(b) はそのキーワードによって選ばれた図 3 のコーディングパターンを基にして生成したコード片を，図 4(a) のソースコードに挿入した結果である．図 4(b) の黒枠で囲まれている部分が，図 3 のコーディングパターンから生成したコード片である．

本手法のコーディングパターン検索方法と生成したコード片が妥当であることを検証するために，評価実験を行った．検索方法の妥当性を検証するため行った実験では，実験対象として 4 つのオープンソースソフトウェアから得た 180 件から 7500 件ほどの各コーディングパターン集合を使用した．そして，実験対象の中に存在するコーディングパターンを，そのコーディングパターンに出現する単語によって取得することが可能であることを確認した．具体的には，コーディングパターンが他のコーディングパターンではほとんど出現しないような単語を含むか，もしくは開発者が多くのキーワードを入力すれば，コーディングパターン集合から目的のコーディングパターンを取得することが可能であることを確かめた．また本手法で生成したコード片が妥当であることを検証した．具体的には，オープンソースソフトウェアから 11 個のコーディングパターンを選択し，選択したコーディングパターンから生成したコード片と，そのコーディングパターンが出現するソースコードを比較した．その

結果，特定のケースを除いて両者の間に大きな違いがなかったことを確認した．

以降，2 章ではコーディングパターンについて説明し，3 章で提案手法について述べる．4 章では提案手法をプログラミング言語 Java に適用した際の実装について述べ，5 章では本手法に対して行った評価実験とその結果についての考察を行う．6 章では関連研究について紹介する．最後に 7 章で本研究のまとめと今後の課題について述べる．

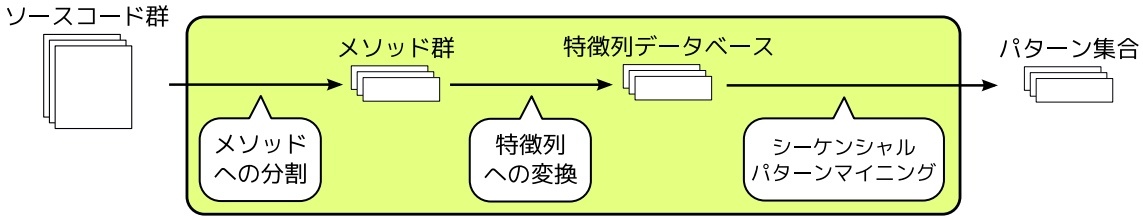


図 5: パターンマイニング全体の手順

2 コーディングパターン

本研究では，既存のソースコード群に頻出するコード片，すなわちコーディングパターンを用いてコード補完を行う．

コーディングパターンとは，複数のソースコードに共通して出現するメソッド呼び出しと制御構造要素の列を抽出したものである．メソッド呼び出しは，そのメソッドをもつクラスの完全修飾名，メソッドの名前，メソッドの戻り値の型，メソッドの各引数の名前と型から成り，制御構造要素は，IF，ELSE，END-IF，LOOP，END-LOOP のいずれかである．パターンの構成要素はソースコード中に連続で出現する必要はないが，パターン中での要素の順序は，ソースコード中での出現順序と一致していなければならない．我々の研究グループでは，パターンをソースコード群から得るパターンマイニング手法を提案している [9, 19]．

図 3 は図 2 のソースコードから得たパターンの例である．このパターンは，jEdit4.3pre11 に実装されている，テキストの変更ができないときは音を鳴らしてユーザーに通知するという処理を表している．

2.1 コーディングパターンマイニング

本研究では，ソースコード群からパターン集合を得るパターンマイニング手法 [9, 19] により，コード補完に用いるパターンを取得する．パターンマイニングの全体の手順を図 5 に示す．また，パターンマイニングの具体的な手順は下記の通りである．

1. ソースコード群をメソッド単位に分割する．
2. 分割した各メソッドを，メソッド呼び出しと制御構造要素の列へと変換し，それをデータベースへ保存する．以降，メソッド呼び出しと制御構造要素の列を特徴列とよび，各メソッドから変換した特徴列から成るデータベースを特徴列データベースとよぶ．
3. 構築した特徴列データベースに対して PrefixSpan アルゴリズム [13] を用いたシーケンシャルパターンマイニングを適用し，パターン集合を抽出する．

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

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

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

図 6: 制御構造要素の変換規則

以降では，特徴列データベースを構築するために，分割したメソッドを特徴列へと変換する方法と，構築した特徴列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を適用し，パターン集合を抽出する方法について説明する．

2.1.1 特徴列への変換

[9, 19] のパターンマイニング手法では，1 つのメソッドを対応する特徴列へと変換する．メソッドの各要素を，順に下記のように特徴列の要素へ変換することで，メソッドから特徴列への変換を行う．

- メソッドの呼び出し

  呼び出されているメソッドを，特徴列の要素となるメソッド呼び出しへと変換する．

- if 文，while 文，for 文

  図 6 の規則に従って変換を行う．この規則は，制御フローに合わせて特徴列の要素となるメソッド呼び出しを並べている．

- その他

  特徴列の要素への変換は行わない．例えば，算術演算や変数の代入・参照，定数の出現などは特徴列に含まない．

図 7 はメソッドから特徴列への変換の例である．ただし，この例では簡単のために特徴列でのメソッド呼び出しをメソッド名だけで表現している．

2.1.2 シーケンシャルパターンマイニング

前節の方法によって得た特徴列から成る特徴列データベースに対して，PrefixSpan[13] アルゴリズムを用いたシーケンシャルパターンマイニングを適用することで，パターン集合を得ることができる．シーケンシャルパターンマイニングは，特徴列データベースを構成する各特徴列から頻出する部分列をパターンとして抽出する手法である．

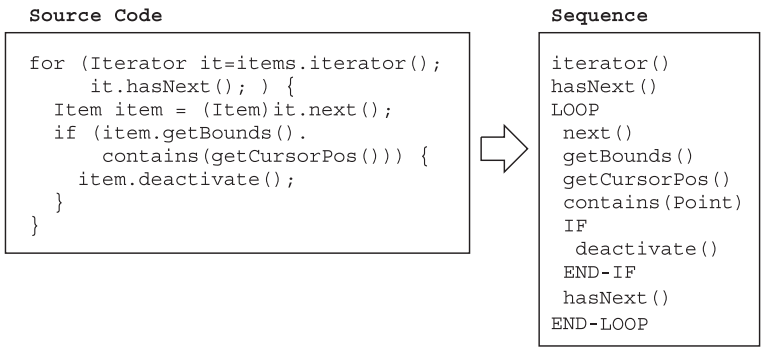


図 7: 特徴列への変換例

PrefixSpan アルゴリズムは，特徴列データベースとパターンの最小出現回数 min_sup を入力とし，与えられた特徴列データベースを構成する特徴列の部分列のうち，min_sup 回以上出現する部分列の集合をパターン集合として出力する．PrefixSpan アルゴリズムの手順は下記の通りである．

1. 特徴列データベースにおける各特徴列の要素の出現回数を数え上げ，出現回数が min_sup 回以上の要素を長さ 1 のパターンとする．

2. 下記の操作によって長さ k の各パターン p に対して下記の操作を適用し，長さ $k+1$ のパターンの集合を得る．長さ $k+1$ のパターンが得られなければ，アルゴリズムを終了する．

   (a) パターン p に対する射影データベースを構築する．p に対応する射影データベースを構築するには，まず特徴列データベースの各特徴列から p を部分列として含む特徴列を選択し，選択した特徴列から p の全ての要素が最初に出現した時点までの要素を取り除く．そして，取り除いた後の特徴列のうち，その長さが 1 以上である特徴列を射影データベースを構成する要素とする．

   (b) 射影データベースを構成する各特徴列の各要素の出現回数を数え上げ，そのうち，出現回数が min_sup 回以上であった要素を p の末尾に付け加えたパターンを，新しい長さ $k+1$ のパターンとする．

3. 1，2 の方法によって得た全てのパターン集合の和集合を出力とする．

図 8 は特徴列データベース $\{(a,b,c),(a,c,d),(c,b,a),(a,a,b)\}$ に対してシーケンシャルパターンマイニングを適用し，出現回数 2 回以上で長さ 1 以上のパターン集合を抽出している様子である．具体的な手順は下記の通りである．以降では長さ k のパターンの集合を P_k，

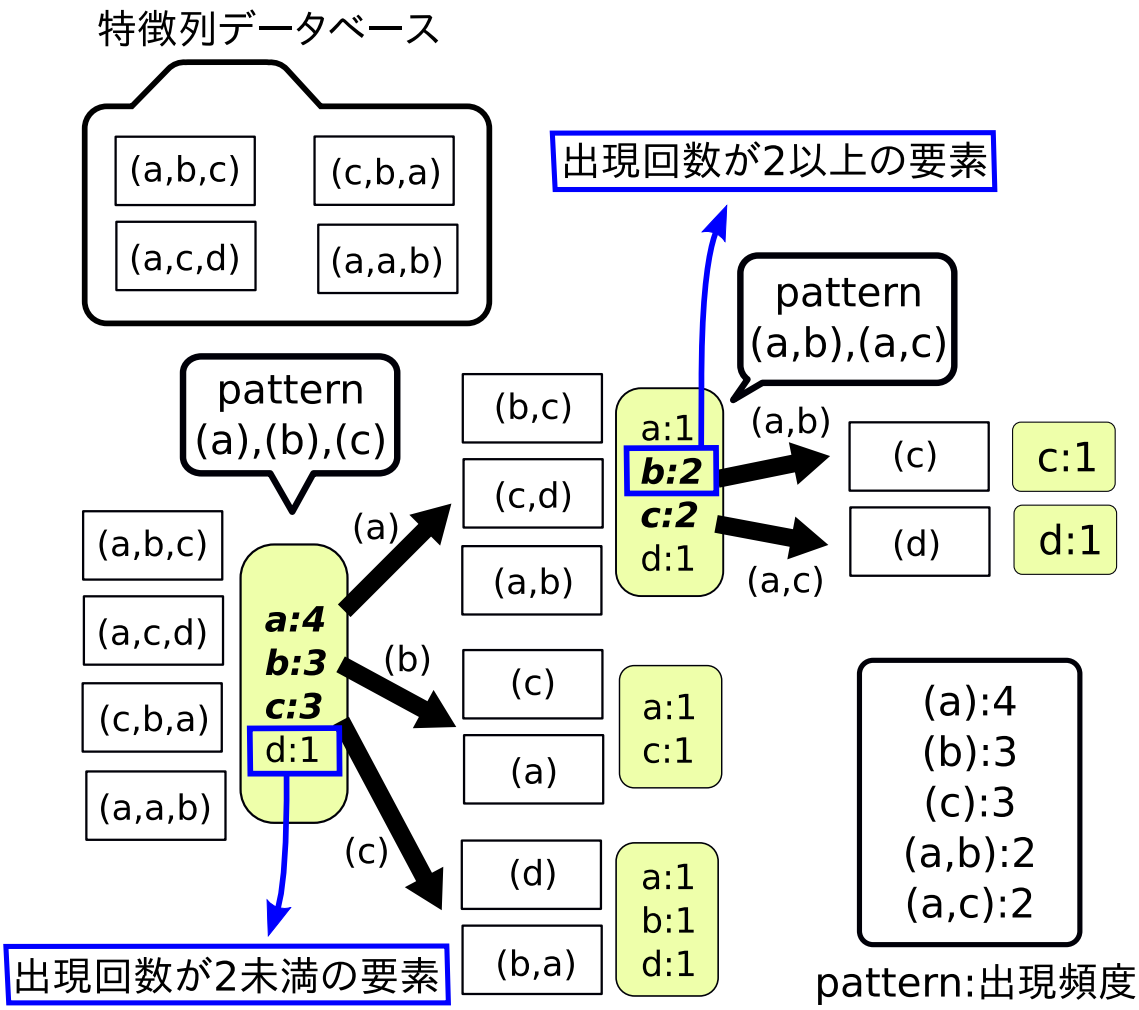


図 8: $min\_sup = 2$，長さ 1 以上のパターン抽出の例

パターン p に対応する射影データベースを $Proj_p$ と表し，図 8 の特徴列データベースを P_0 と表す.

1. P_0 を構成する特徴列の各要素のうち，2 回以上出現する要素は a, b, c なので，$P_1 = \{(a), (b), (c)\}$ となる.

2. P_1 の各パターンから長さ 2 のパターンを下記のようにして得る.

   - $a \in P_1$

     $Proj_{(a)} = \{(b, c), (c, d), (a, b)\}$ で，$Proj_{(a)}$ を構成する特徴列の各要素のうち，2 回以上出現する要素は b, c なので，(a) の末尾にこれらの要素を加えた $(a, b), (a, c)$ が長さ 2 のパターンとなる.

- $b \in P_1$

  $Proj_{(b)} = \{(c), (a)\}$ で，$Proj_{(b)}$ を構成する特徴列の各要素のうち，2 回以上出現する要素はないため，b からは長さ 2 のパターンは発生しない．

- $c \in P_1$

  b と同様に c からは長さ 2 のパターンは発生しない．

以上の結果から $P_2 = \{(a, b), (a, c)\}$ である．

3. P_2 の各パターンから長さ 3 のパターンを下記のようにして得る．

   - $(a, b) \in P_2$

     $Proj_{(a,b)} = \{(c)\}$ で，$Proj_{(a,b)}$ を構成する特徴列の各要素のうち，2 回以上出現する要素はないため，(a, b) からは長さ 3 のパターンは発生しない．

   - $(a, c) \in P_2$

     (a, b) と同様に (a, c) からは長さ 3 のパターンは発生しない．

   以上の結果から $P_3 = \emptyset$ なので，長さ 4 以上のパターンの集合は抽出しない．

4. P_1, P_2 より，P_0 の特徴列から抽出した出現回数 2 回以上，長さ 1 以上のパターンの集合は $\{(a), (b), (c), (a, b), (a, c)\}$ である．

3　提案手法

本研究では，ソフトウェアのパターン情報を基に，開発者が編集しているソースコードの補完を行うコード補完手法を提案する．本手法は，入力としてパターン集合，開発者が編集しているソースコード，開発者が入力したキーワードを受け取る．そして，入力のキーワードに関連したパターンに基づいて，入力のソースコードに適合したコード片を出力する．具体的には，まずパターン集合から，開発者が入力したキーワード集合 IK に最も合致する上位 k 個のパターンを選択する．そして，選択したパターンと開発者が編集しているソースコードのコンテキストを用いて，コード片を生成する．

3.1　コーディングパターンの検索

一般にソフトウェアからは複数のパターンを得ることができ，その中には開発者が目的とする機能とは関連のないパターンも含まれる．例えば，図 3 のパターンはテキストに変更を加える処理を行う際に，テキストが編集できなければ音を鳴らしてユーザーに通知するという処理を表しているが，jEdit4.3pre11 には他のパターンとして，例えばログファイルへ編集しているファイルの絶対パスを出力するパターンが確認された．本手法では開発者にとって有用なコード片を生成するために，パターン集合に出現する単語の重み付け，開発者からの入力キーワード集合 IK，各メソッド呼び出し m の評価値 $MScore(m)$ を用いて，パターンの検索を行う．パターンの検索は，各パターン p の評価値 $PScore(p)$ を求め，最も評価値の大きい k 個のパターンを選択する．選択したパターンはコード片の生成で利用する．

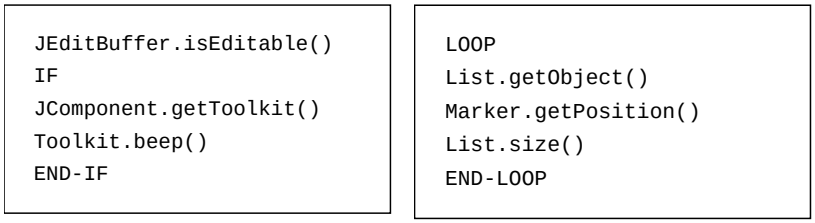
以降では，まず単語の重み付けについて説明し，次にパターンを構成するメソッド呼び出し m の評価値 $MScore(m)$ を求める方法について述べる．最後にパターン p の評価値 $PScore(p)$ を求める方法について説明する．

3.1.1　単語の重み付け

パターン集合に出現する単語には set や get のように多数のパターンに出現する単語と，editable のように限られたパターンにのみ出現する単語が存在する．本手法ではパターン中の出現回数が少ない単語ほどパターンやメソッド，IK の特徴を表現する重要な単語であるとみなして単語の重み付けを行う．

パターンに出現する単語は，パターンを構成するメソッド呼び出しに出現する単語から求める．メソッド呼び出し m から得られる単語の集合 $MW(m)$ は，メソッド呼び出し m の構成要素をキャメルケース，アンダースコア，数を区切りとして分割して得る．

例として図 9 のメソッド呼び出しに出現する単語の集合を求める．メソッド呼び出しのメソッド名とクラス名から単語を取得すると，各メソッド呼び出しに対応する単語集合は次の



(a) (b)

図 9: コーディングパターンの例

ようになる．

$$\begin{aligned} MW(\text{JEditBuffer.isEditable()}) &= \{\text{j}, \text{edit}, \text{buffer}, \text{is}, \text{editable}\} \\ MW(\text{List.getObject()}) &= \{\text{list}, \text{get}, \text{object}\} \\ MW(\text{List.size()}) &= \{\text{list}, \text{size}\} \\ MW(\text{Marker.getPosition()}) &= \{\text{marker}, \text{get}, \text{position}\} \end{aligned}$$

パターン p に出現する単語の集合 $PW(p)$ は，p を構成する各メソッド呼び出しから得られる単語集合の和集合である．$PW(p)$ を式で表すと次のようになる．

$$PW(p) = \bigcup_{m \in PM(p)} MW(m)$$

単語の重みは TF-IDF(Term Frequency - Inverse Document Frequency)[15] を用いて決定する．ただしパターン集合を 1 つの文書とみなすため，IDF 項は定数とみなす．全てのパターン中に登場する全ての単語を個別に数えた場合の総数を N とし，全てのパターンの中で単語 w が出現する数を n_w とする．このとき，単語 w の重み $Weight(w)$ を次のように定義する．

$$Weight(w) = \begin{cases} \log(N/n_w) & (n_w > 0) \\ 0 & (n_w = 0) \end{cases}$$

例として図 9 にある 2 つのパターンに出現する単語について重み付けを行う．ただし簡単のために，対象となる単語はメソッド名とクラス名に出現するものだけとする．このとき，N と単語の重みは次のようになる．

$$\begin{aligned} N &= 19 \\ Weight(\text{editable}) &= \log(19/1) \\ Weight(\text{buffer}) &= \log(19/1) \end{aligned}$$

$$
\begin{aligned}
Weight(\text{list}) &= \log(19/2) \\
Weight(\text{get}) &= \log(19/3) \\
Weight(\text{property}) &= 0
\end{aligned}
$$

3.1.2 メソッド呼び出しの評価

本節では単語の重みと入力キーワード集合 IK を用いて，パターンを構成するメソッド呼び出し m の評価値 $MScore(m)$ を求める方法について説明する．

メソッド呼び出し m の評価値 $MScore(m)$ は，IK と m に出現する単語の集合 $MW(m)$ に共通する単語の重みの総和である．$MScore(m)$ を式で表すと次のようになる．

$$
MScore(m) = \sum_{w \in IK \cap MW(m)} Weight(w)
$$

例として図 9 のメソッド呼び出しの評価値を求める．3.1.1 節の結果から，IK ={list, buffer, editable} としたときのメソッド呼び出しの評価値は次のように求められる．

$$
\begin{aligned}
MScore(\text{JEditBuffer.isEditable()}) &= Weight(\text{buffer}) + Weight(\text{editable}) = 2\log(19/1) \\
MScore(\text{List.getObject()}) &= Weight(\text{list}) = \log(19/2) \\
MScore(\text{List.size()}) &= Weight(\text{list}) = \log(19/2) \\
MScore(\text{Marker.getPosition()}) &= 0
\end{aligned}
$$

3.1.3 パターンの評価

本節では単語の重み，入力キーワード集合 IK，各メソッド呼び出し m の評価値 $MScore(m)$ を用いてパターン p の評価値 $PScore(p)$ を求める方法について説明する．パターン p に対する評価値 $PScore(p)$ は，p を構成する各メソッド呼び出しの評価値の平均と，IK に含まれ p に含まれないキーワードの重みの総和をペナルティとして与えたものである．

パターン p に含まれるメソッド呼び出しの列を $PM(p)$ とする．すなわち，あるパターン p で同じメソッドを複数回呼び出しているような場合，各メソッド呼び出しは個別に $PM(p)$ の要素となる．まず，パターン p に対するペナルティ $Penalty(p)$ を，p に出現する単語の集合 $PW(p)$ を用いて次のように定義する．

$$
Penalty(p) = 1 + \sum_{w \in IK \setminus PW(p)} Weight(p)
$$

このとき，パターン p の評価値 $PScore(p)$ を次のように定義する．

$$
PScore(p) = \begin{cases} 0 & if\ IK \cap PW(p) = \emptyset \\ \frac{\sum_{m \in PM(p)} MScore(m)}{|PM(p)|} \times \frac{1}{Penalty(p)} & otherwise \end{cases}
$$

例として図 9 のパターンの評価値を求める．図 9(a) のパターンを p_1，図 9(b) のパターンを p_2 とし，IK ={list,buffer,editable} とする．3.1.1 節の結果から p_1, p_2 のペナルティは次のようになる．

$$\begin{aligned} Penalty(p_1) &= 1 + Weight(\text{list}) = 1 + \log(19/2) \\ Penalty(p_2) &= 1 + Weight(\text{buffer}) + Weight(\text{editable}) = 1 + 2\log(19/1) \end{aligned}$$

ペナルティと 3.1.2 節の結果を用いると，p_1, p_2 の評価値は次のようになる．

$$\begin{aligned} PScore(p_1) &= \frac{MScore(\text{JEditBuffer.isEditable()})}{3} \times \frac{1}{Penalty(p_1)} \\ &= \frac{2\log(19/1)}{3(1+\log(19/2))} \\ PScore(p_2) &= \frac{MScore(\text{List.getObject()}) + MScore(\text{List.size()})}{3} \times \frac{1}{Penalty(p_2)} \\ &= \frac{2\log(19/2)}{3(1+2\log(19/1))} \end{aligned}$$

以上の方法を用いて，パターン集合の各パターンの評価値を決定する．そして最も評価値の大きい上位 k 個のパターンから，開発者へ推薦するコード例に挿入するコード片を生成する．

3.2 コード片の生成

前節で述べた検索方法によって取得したパターンと，開発者が編集しているソースコードのコンテキストを用いてコード片の生成を行う．本手法ではパターン 1 つに対し 1 つのコード片を生成する．そして，生成したコード片を開発者が編集しているソースコードに挿入し，開発者に推薦する．

パターンを構成するメソッド呼び出しの戻り値は，その後メソッド呼び出しの receiver object や実引数，制御構造要素の条件式，もしくは生成したコード片以降のソースコードで利用されることがある．例えば，図 2 に示したソースコードでは，いずれも JEditBuffer.isEditable() の戻り値を if 文の条件式に利用し，また JComponent.getToolkit() の戻り値に対してメソッド beep() を呼び出していることがわかる．本手法では，パターンの各メソッド呼び出しの戻り値をその後上記で述べたような用途で利用できるようにするために，コンテキストを用いる．

コンテキスト CTX は型 t と変数名 v のペア (t, v) を要素とする集合で，同じ型をもつ要素は CTX に高々1 つしか存在しない．すなわち，CTX は次の制約を満たす集合である．

$$\forall (t, v), (t', v') \in CTX.t = t' \Rightarrow v = v'$$

要素番号	パターンの要素
1	JEditBuffer.isEditable()
2	IF
3	JComponent.getToolkit()
4	Toolkit.beep()
5	END-IF

(a) コーディングパターン

```
public void editBuffer() {
  JEditBuffer buf = this.getBuffer();

 [A code snippet will be inserted here]
}
```

(b) ソースコード

図 10: コード片の生成例で用いるコーディングパターンとソースコード

本手法では基となるパターンの構成要素を順に具体化することでコード片を生成する．具体化の際に，メソッド呼び出しや制御構造要素の条件式として型 t の変数が必要になったとき，コンテキスト CTX に型 t をもつ要素 (t, v) が存在すれば変数 v を用いる．CTX にそのような要素が存在しなければ，生成途中のコード片でその都度型 t の変数を新しく宣言する．このとき宣言する変数の名前はユニークなものとする．そして，新しく宣言した型 t と変数名 v を対とする要素 (t, v) を CTX へ加える．またパターンの各メソッド呼び出しが戻り値を返すときは，その度にユニークな名前の変数を宣言し，宣言した変数へ戻り値を格納するようなコードを生成途中のコード片へ加える．そして，その型と変数名の対を CTX へと加える．コード片生成開始直後の CTX は，生成したコード片を挿入する位置で利用可能な変数とその型のペアの集合である．ただし，型 t と対となっている変数 v は，生成したコード片を挿入する位置に最も近い位置で宣言された変数となっている．例えば，プログラミング言語 Java に適用した場合では，フィールドとメソッドの仮引数が同じ型であればメソッドの仮引数に対応する変数名を CTX に加え，メソッドの仮引数と局所変数では局所変数名を CTX に加える．

例として図 10 のパターンとソースコードを用いてコード片を生成する．コード片生成開始直後のコンテキスト CTX は

$$CTX = \{(\mathrm{JEditBuffer}, \mathrm{buf})\}$$

である．次に，コード片を生成するために図 10(a) のパターンの各要素を具体化していく．表 1 はパターンの各要素と，要素を具体化するときに出力するコード片に加えるソースコードの断片の関係を表している．また，表 2 はパターンの各要素を具体化した直後の CTX を示している． ただし，表 2 では CTX 中の型のうち，パッケージ名を省略している．パターンの各要素の具体化が完了すると，生成したコード片は図 11 のようになる．

コード片の生成が完了すると，生成したコード片を開発者が編集しているソースコードに挿入したコード例を，開発者へ提示する．

```
boolean var0;
var0 = buf.isEditable();
if (var0) {
  java.awt.Toolkit var1;
  javax.swing.JComponent var2;
  var1 = var2.getToolkit();
  var1.beep();
}
```

図 11: 図 10(a) のパターンから生成したコード片

表 1: 図 10(a) のパターン要素を具体化するときに生成するコード片

パターンの要素番号	生成コード片
1	boolean var0;
1	var0 = buf.isEditable();
2	if (var0) {
3	java.awt.Tookit var1;
3	javax.swing.JComponent var2;
3	var1 = var2.getToolkit();
4	var1.beep();
5	}

表 2: 図 10(a) のパターン要素の具体化直後のコンテキスト

パターンの要素番号	CTX
1	{(JEditBuffer, buf),(boolean, var0)}
2	{(JEditBuffer, buf),(boolean, var0)}
3	{(JEditBuffer, buf),(boolean, var0),(Toolkit,var1),(JComponent, var2)}
4	{(JEditBuffer, buf),(boolean, var0),(Toolkit,var1),(JComponent, var2)}
5	{(JEditBuffer, buf),(boolean, var0)}

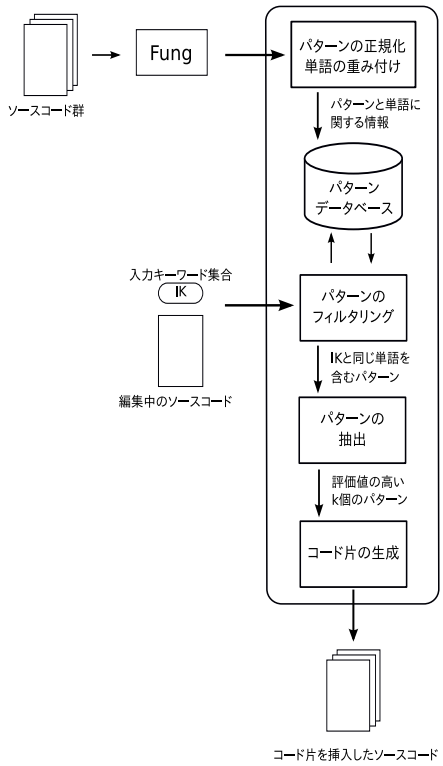


図 12: システム構成図

4 実装

本研究では提案手法をプログラミング言語 Java へ適用するために，Eclipse[2] プラグインを実装した．図 12 は開発したツールの全体構成図である．パターン集合はコーディングパターン検出ツール Fung[3] の出力を利用している．単語の重みはパターン集合だけで決定することができるため，本ツールではパターン集合が与えられた時点で単語の重みを決定している．

パターンのフィルタリングでは，評価値を求めるパターンを，評価値が 0 より大きい，すなわち IK 中の単語が出現するパターンに制限している．これは評価値が 0 のパターンはキーワード IK と無関係だと考えられるためである．このとき，パターンの数は膨大であるため全てのパターンに対してキーワードの出現をテストするとフィルタリングに多くの時間を費やすことになる．そこで本ツールではトライ木 [10] によってキーワードを含むパターンを選択することで，フィルタリングを効率的に行っている．

フィルタリング後の各パターンの評価値を求め，評価値が最も高い上位 k 個のパターンを選択する．本ツールでは $k = 3$ としている．次に選択したパターンと開発者が編集している

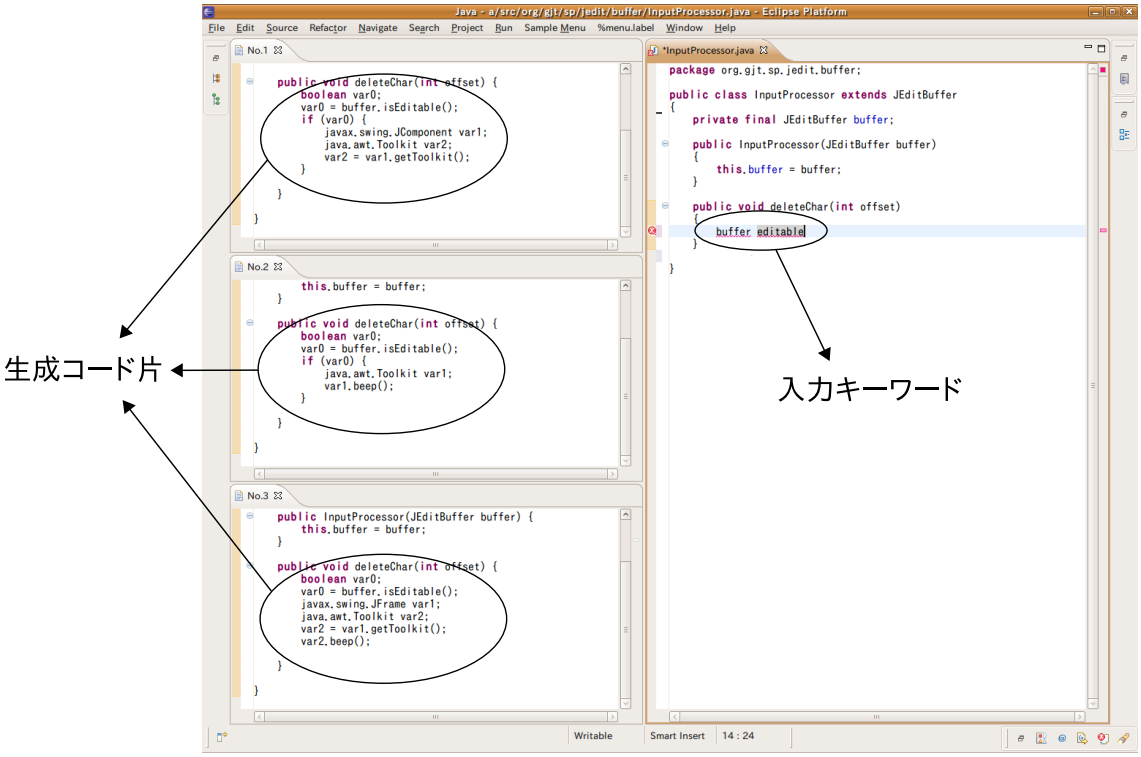


図 13: ツールのスクリーンショット

ソースコードのコンテキストからコード片を生成する．コード片を生成する際に用いるコンテキストは Eclipse の Java Develpment Tools に含まれる AST(abstract syntax tree) 生成器を利用して取得する．最後に，生成したコード片を編集中のソースコードへ挿入したコード例を開発者へ提示する．ただし，Fung で型が解決できないよう場合は，型名を unknown として扱っている．

図 13 に，実装したツールのスクリーンショットを示す．選択した 3 つのパターンから生成したコード片を挿入したコード例を，ウィンドウ左側の小さなテキストエディタ領域に表示している．

5　評価実験

4 章で述べたツールを用いて評価実験を行い，次の項目について評価した．

- パターン評価式の妥当性

また，次の項目について調査を行った．

- 生成したコード片の妥当性

以降では各評価実験の方法と結果，そして結果に対する考察を述べる．

5.1　パターン評価式の妥当性

本実験ではパターン評価式の妥当性を検証するために，あるパターン集合の「特徴的なパターン」p について，p に出現する単語をキーワードとしてパターンの検索を行ったときの p の評価値についての調査を行った．「特徴的なパターン」とは「特徴的な単語」を含むパターンのことであり，「特徴的な単語」とは重みの大きい単語のことである．

5.1.1　実験方法

検索に使用する「特徴的な単語」の集合 W_e は，コーディングパターン検出ツール Fung の出力となるパターン集合 P_o から得た単語集合 W_o のうち，重みの大きい順で順位付けしたときの上位 $\frac{9}{10}$ または $\frac{1}{2}$ に含まれる単語の集合である．以降，上位 $\frac{9}{10}$ に含まれる単語の集合を $W_{9/10}$，上位 $\frac{1}{2}$ に含まれる単語の集合を $W_{1/2}$ と表す．検索対象となるパターンの集合 P_e は，P_o のパターンのうち W_e に含まれる単語が 1 つ以上出現するパターンの集合である．パターン p に出現する単語の集合 $PW(p)$ を用いると，P_e は次のように定義できる．

$$P_e = \{p \mid p \in P_o \ \&\ PW(p) \cap W_e \neq \emptyset\}$$

以降では $W_e = W_{1/2}$ であるときの P_e を $P_{1/2}$ と表し，$W_e = W_{9/10}$ であるときの P_e を $P_{9/10}$ と表す．$p \in P_e$ である各パターン p に対して検索を行うときに用いるキーワード集合は，下記の 2 通りの方法から得る．

1. 空集合を除く $PW(p) \cap W_e$ の任意の部分集合．
2. 空集合を除く，要素数が 5 以下である $PW(p) \cap W_e$ の任意の部分集合．

以上の条件を満たすキーワード集合 $IK_1, IK_2, \cdots, IK_n$ を列挙する．そして，列挙した各キーワード集合に対してパターン集合 P_o の各パターンに対して評価値の計算を行い，そのときのパターン p の順位について調査を行った．

5.1.2 実験結果

本節では上記で述べた実験方法を 4 つのオープンソースソフトウェアに適用した結果について説明する．実験に使用したソフトウェアと，ソフトウェアから得られたパターンの総数 $|P_o|$ と $|P_o|$ から得られた単語の総数 $|\bigcup_{p \in P_o} PW(p)|$ を表 3 に示す．$W_e = W_{9/10}, W_{1/2}$ としたときの特徴的な単語の数 $|W_e|$，特徴的なパターンの数 $|P_e|$ を表 4 に示す．また，特徴的な単語集合 W_e とキーワード集合の取得方法ごとに表 5 のように実験条件を設定すると，各実験条件で用いたキーワード集合 IK の要素数の最大値 $|IK|_{max}$ と平均値 $|IK|_{ave}$ は表 6 のようになった．

それぞれの実験条件でのパターンの順位とその順位になったパターン数の累積度数の関係を図 14，15，16，17 に示す．また，それぞれの実験条件で P_e の全てのパターンに対して行なった実験回数の総数 P_{exp} と，そのうちパターンの順位が上位 3 位以内であった (Eclipse 上でコード例が表示される) パターンの総数 P_3，そして P_{exp} と P_3 の比を表 7 に示す．

表 3: 実験に使用したソフトウェア

ソフトウェア	$\|P_o\|$	$\|\cup_{p \in P_o} PW(p)\|$
jEdit4.3pre11	7559	216
JHotDraw7.0.9[5]	7813	321
ASM3.2[1]	678	132
PDF Render1.3[6]	186	91

表 4: 特徴的な単語の数 $|W_e|$ と検索対象のパターンの数 $|P_e|$

(a) $W_e = W_{9/10}$

ソフトウェア	$\|W_e\|$	$\|P_e\|$
jEdit4.3pre11	195	6003
JHotDraw7.0.9	289	3355
ASM3.2	119	668
PDF Render1.3	82	186

(b) $W_e = W_{1/2}$

ソフトウェア	$\|W_e\|$	$\|P_e\|$
jEdit4.3pre11	108	124
JHotDraw7.0.9	160	261
ASM3.2	66	61
PDF Render1.3	45	34

5.1.3 考察

表 6(c)，7(c) と表 6(d)，7(d) から，本手法のパターン検索方法は $P_{1/2}$ のパターンについては検索の際に使用したキーワードの数に関わらず，目的のパターンに出現する単語を検索のキーワードに使うことにより，そのパターンを高い確率で上位 3 位以内のパターンとして取得できることがわかった．また表 6(a)，7(a) から，重みの小さな単語のみを含むパターンを検索する場合は，そのパターンを取得するために必要なキーワードの数が大きくなることがわかる．表 6(b)，7(b) からは，$P_{9/10}$ のパターンを検索するために使用するキーワードの数を減らすと，目的のパターンを取得することが困難になることが確認された．パターンに出現する単語の重みが小さいということは，そのパターンに出現する単語がより多くのパ

表 5: 実験条件の割り当て

実験番号	W_e	キーワード集合の取得方法
実験 1	$W_{9/10}$	1
実験 2	$W_{9/10}$	2
実験 3	$W_{1/2}$	1
実験 4	$W_{1/2}$	2

表 6: キーワード集合 IK の要素数の最大値 $|IK|_{max}$ と平均値 $|IK|_{ave}$

(a) 実験 1

ソフトウェア	$\|IK\|_{max}$	$\|IK\|_{ave}$
jEdit4.3pre11	15	5.971
JHotDraw7.0.9	22	8.974
ASM3.2	18	7.764
PDF Render1.3	12	5.585

(b) 実験 2

ソフトウェア	$\|IK\|_{max}$	$\|IK\|_{ave}$
jEdit4.3pre11	5	3.618
JHotDraw7.0.9	5	4.297
ASM3.2	5	4.196
PDF Render1.3	5	4.075

(c) 実験 3

ソフトウェア	$\|IK\|_{max}$	$\|IK\|_{ave}$
jEdit4.3pre11	9	3.471
JHotDraw7.0.9	7	2.338
ASM3.2	9	3.464
PDF Render1.3	5	2.125

(d) 実験 4

ソフトウェア	$\|IK\|_{max}$	$\|IK\|_{ave}$
jEdit4.3pre11	5	2.977
JHotDraw7.0.9	5	2.276
ASM3.2	5	3.115
PDF Render1.3	5	2.125

ターンに出現するということである．そのためパターン検索に用いるキーワードが少なくなると，同じ単語が多く出現するパターン間の違いを求めることが困難となる．

以上の結果より，目的のパターンが重みの大きな単語を含むか，もしくは開発者が多くのキーワードを入力すれば，本手法は入力キーワード集合に関連したパターンを検索することが可能であることが確認できた．しかし，目的のパターンに重みの小さな単語しか出現せず，開発者が少数 (本実験ではこの値を 5 とした) のキーワードしか与えないような場合には，本

表 7: 各実験の対象となったパターンと上位パターンの総数

(a) 実験 1

ソフトウェア	P_{exp}	P_3	P_3/P_{exp}
jEdit4.3pre11	363231	242810	0.6685
JHotDraw7.0.9	20821929	14321249	0.6878
ASM3.2	1167277	1425491	0.8189
PDF Render1.3	44213	28034	0.6341

(b) 実験 2

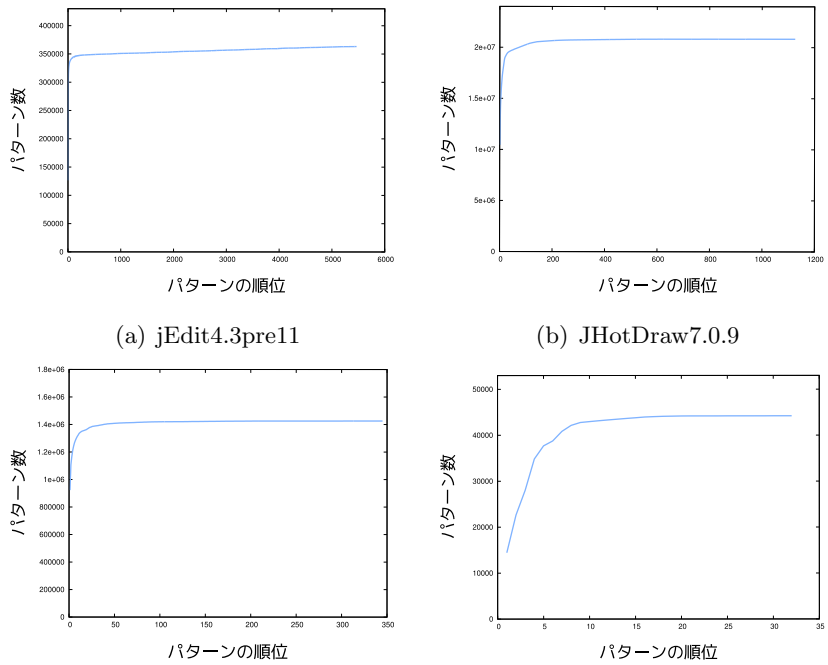
ソフトウェア	P_{exp}	P_3	P_3/P_{exp}
jEdit4.3pre11	147513	64473	0.4371
JHotDraw7.0.9	1932176	592641	0.3067
ASM3.2	236853	117652	0.4967
PDF Render1.3	21197	11684	0.5512

(c) 実験 3

ソフトウェア	P_{exp}	P_3	P_3/P_{exp}
jEdit4.3pre11	1158	1132	0.9775
JHotDraw7.0.9	1605	1302	0.8112
ASM3.2	1475	1429	0.9688
PDF Render1.3	327	231	0.7604

(d) 実験 4

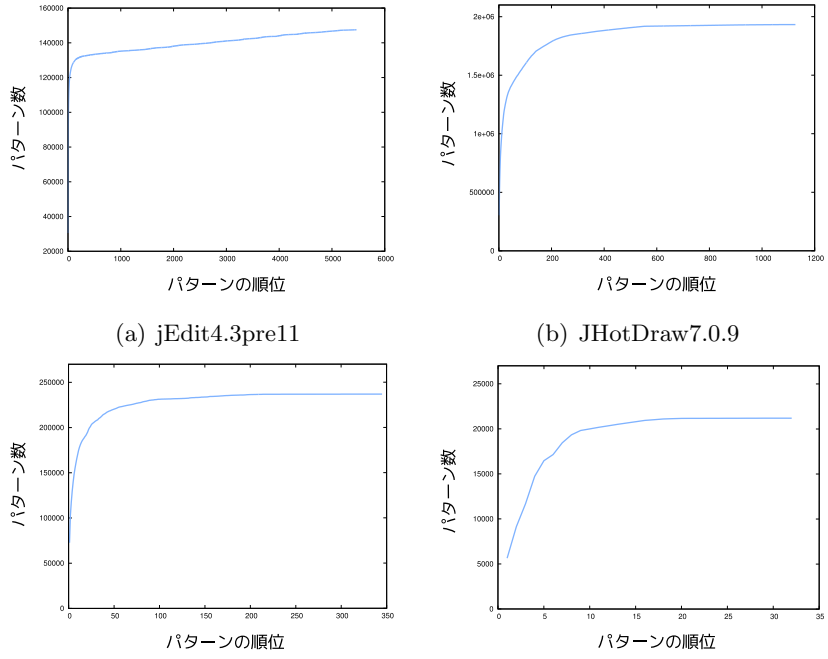
ソフトウェア	P_{exp}	P_3	P_3/P_{exp}
jEdit4.3pre11	991	965	0.9738
JHotDraw7.0.9	1579	1276	0.8081
ASM3.2	1317	1271	0.9651
PDF Render1.3	327	231	0.7604



(a) jEdit4.3pre11　(b) JHotDraw7.0.9

(c) ASM3.2　(d) PDF Render1.3

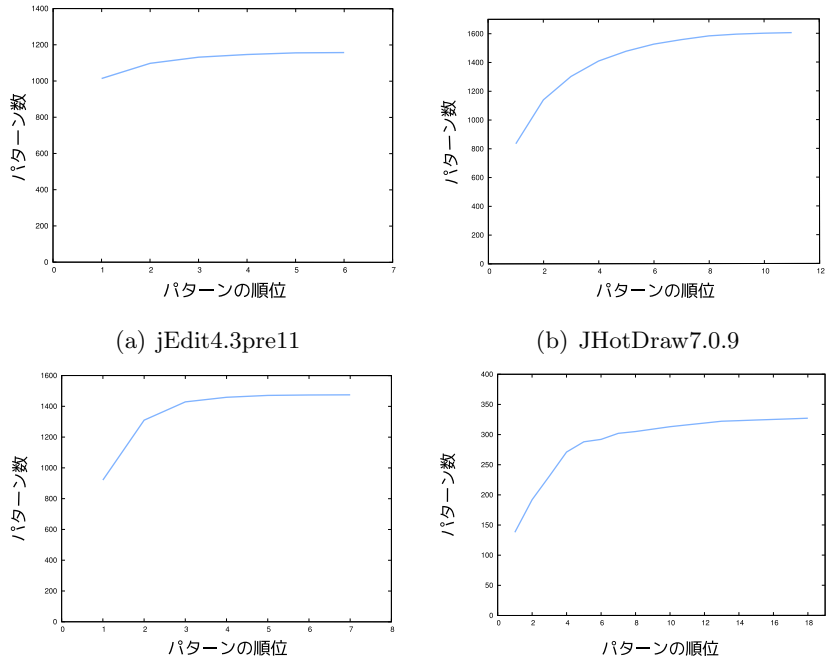
図 14: 実験 1 でのパターンの評価値順位とパターン数の累積度数の関係



(a) jEdit4.3pre11　(b) JHotDraw7.0.9

(c) ASM3.2　(d) PDF Render1.3

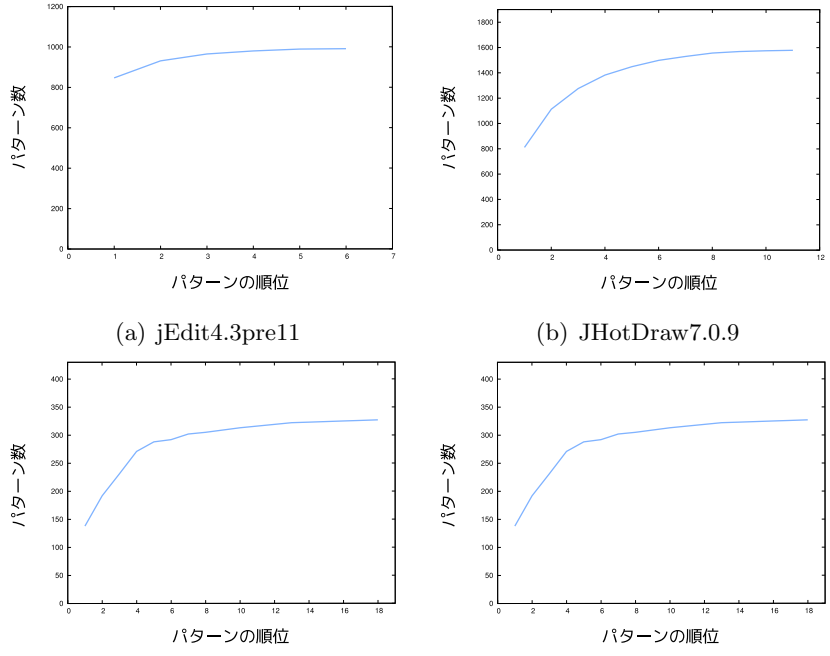
図 15: 実験 2 でのパターンの評価値順位とパターン数の累積度数の関係



(a) jEdit4.3pre11　(b) JHotDraw7.0.9

(c) ASM3.2　(d) PDF Render1.3

図 16: 実験 3 でのパターンの評価値順位とパターン数の累積度数の関係



(a) jEdit4.3pre11　(b) JHotDraw7.0.9

(c) ASM3.2　(d) PDF Render1.3

図 17: 実験 4 でのパターンの評価値順位とパターン数の累積度数の関係

手法のパターン検索方法では目的のパターンを取得することが困難であることがわかった．この問題を解決する手段として，本手法におけるパターンの評価式を改良する以外に，キーワード集合以外の情報を用いたパターン検索方法を考案することが考えられる．パターン検索の際に利用可能なキーワード集合以外の情報については 7 章で述べる．

5.2 生成したコード片の妥当性

3.2 節で述べた方法により生成したコード片が妥当であるかについて調査を行った．

5.2.1 調査方法

本実験では，本手法をプログラミング言語 Java に適用したツールが生成したコード片の妥当性を調査する．方法としては，まず既存のソフトウェアから抽出したパターン集合からランダムにパターンを選択する．そして，選択したパターンが出現するソースコードと，そのソースコードのコンテキストと選択したパターンを用いて生成したコード片を比較し，その違いについて調査を行った．調査対象のソフトウェアは jEdit4.3pre11 で，調査を行ったソースコード上の位置は表 8 の通りである．

表 8: jEdit4.3pre11 での調査対象

クラス名	メソッド名	行番号
org.gjt.sp.jedit.textarea.TextArea	goToPrevFold	3783
org.gjt.sp.jedit.textarea.TextArea	insertTabAndIndent	4396
org.gjt.sp.jedit.textarea.TextArea	tabsToSpaces	4248
org.gjt.sp.jedit.textarea.InputMethodSupport	paintValidLine	90
org.gjt.sp.jedit.BeanShell	classLoaderChanged	815
org.gjt.sp.jedit.gui.DefaultInputHandler	removeKeyBinding	174
org.gjt.sp.jedit.gui.ColorWellButton	init	202
org.gjt.sp.jedit.gui.EditAbbrevDialog	init	113
org.gjt.sp.jedit.gui.FontSelector	init	372
org.gjt.sp.jedit.options.ContextOptionPane	actionPerformed	218
org.gjt.sp.jedit.options.GeneralOptionPane	_init	94

```
Component Box.add(Component)
Component Box.add(Component)
Component Box.add(Component)
Component Box.createGlue()
Component Box.add(Component)
```

(a) パターン

```
Box buttons = ...;
buttons.add(Box.createGlue()); ...
JButton ok = new JButton(...); ...
buttons.add(ok);
buttons.add(Box.createHorizontalStrut(...));
...
buttons.add(Box.createGlue());
```

(b) ソースコード

```
Box buttons = ...;
Component var0;
var0 = buttons.add(buttons);
Component var1;
var1 = buttons.add(var0);
Component var2;
var2 = buttons.add(var1);
Component var3;
var3 = buttons.createGlue();
Component var4;
var4 = buttons.add(var3);
```

(c) 生成したコード片

図 18: 変数の使用方法が異なるケース 1 の例

5.2.2 調査結果

調査の結果，次の場合を除き本手法が生成したコード片とソースコード上のパターンに該当する部分について，変数の使用方法に違いは見られなかった．変数の使用方法が大きく異なったケースとそれについての考察は下記の通りである．

1. オブジェクトの協調動作が共通の親クラスのメソッドによって行なわれる．

   これは GUI の設定を行う箇所などが該当する．図 18 は，この問題が発生した際のパターン，ソースコード，生成したコード片である．この例では，図 18(b) のソースコードではメソッド Box.add(Component) の戻り値はその後利用していないのに対し，図 18(c) のコード片ではメソッド Box.add(Component) の戻り値をその後実引数として利用している．

   この問題は，パターンが出現するソースコード上でのデータフローを利用したり，メソッドの receiver object のクラスとメソッドを定義しているクラス，もしくはメソッドの実引数と仮引数のクラスの継承関係に関するルールを取得することである程度対応することができると考えられる．しかし，例えば複数の同じクラスのオブジェクトが存在しており，その使用方法に差がないような場合は，パターンが出現するソース

```
int JEditBuffer.getLineStartOffset()
IF
TextArea.extendSelection(int, int)
ELSE
TextArea.selectNone();
END-IF
```

(a) パターン

```
JEditBuffer buffer; ...
public void goToPrevFold(...) { ...
  int newCaret = buffer.getLineStartOffset ...;
  if (select) {
    extendselection(caret, newCaret);
  }
  else { ...
    selectNone();
  }
  moveCaretPosition(newCaret);
}
```

(b) ソースコード

```
JEditBuffer buffer; ...
public void goToPrevFold(...) { ...
  int var0;
  var0 = buffer.getLineStartOffset ...
  if (select) {
    TextArea var1;
    var1.extendSelection(var0, var0);
  }
  else {
    TextArea var1;
    var1.selectNone();
  }
```

(c) 生成したコード片

図 19: 変数の使用方法が異なるケース 2 の例

コードと同じように変数を使用しているコード片を生成することは困難である．

2. 同じ型の引数を複数とるメソッド呼び出しが出現する．

   このケースの例を図 19 に示す．これは本手法のコード片生成方法で用いるコンテキスト中には，同じ型が高々1 つしか存在しないために発生する．

   この問題の解決する方法として，コード片生成時に用いるコンテキストを拡張することが挙げられる．具体的には，コンテキストが同じ型の変数を複数もつことを許容し，メソッド呼び出しが複数の同じ型を引数とするとき，それぞれ別の変数を使用する．しかしこの方法を用いた場合，参照する変数を決定するために，型以外の情報が必要となる．そのようなときに利用可能な情報については 7 章で述べる．

3. 組込型の値 (String を含む) が登場する．

   組込型の値に対する操作はパターンとして出現せず，また定数としてソースコードに直接記述することができるため，生成したコード片とソースコードで使用する値が異なる箇所が存在した．

   この問題は上記のオブジェクトの協調動作が共通の親クラスのメソッドによって行なわれる場合と同様に，パターンが出現するソースコード上でのデータフローを利用すること

(a) パターン

```
public void removeKeyBinding(...) {
  StringTokenizer st = ...
  while (st.hasMoreTokens()) {
    String s = st.nextToken(); ...
    if (st.hasMoreTokens()) { ...
    } ... } ...
}
```

(b) ソースコード

```
boolean var0;
while (var0) {
  StringTokenizer var1;
  String var2 = var1.nextToken();
  boolean var3 = var1.hasMoreTokens();
}
```

(c) 生成したコード片

図 20: 有効スコープが異なる例

である程度対応することができると考えられる．しかし，SQL による RDB(Relational DataBase) への接続文字列のように，パターンが出現する位置それぞれで異なる定数を使用するような場合，適切なコード片を生成することは困難である．

以上の 3 つの場合を除き，本手法が生成したコード片は妥当だと考えられる．また，ソースコードと生成したコード片には次のような違いが見られた．

- ソースコードと生成したコード片で変数の有効スコープが異なる．

  例えば図 20(a) は図 20(b) のソースコードから抽出したパターンである．このパターンと $StringTokenizer$ オブジェクトが存在しないソースコードを用いてコード片を生成したとき，生成したコード片は図 20(c) のようになる．図 20(b) のソースコードでは，変数 $var1$ に対応する変数 st は while 文の条件式で用いられるため，while ブロックの外側で宣言されている．このように，生成したコード片では内側のスコープで宣言した変数を，開発者のソースコードでは外側のスコープで参照することがある．

- ソースコードには出現しない変数が生成したコード片に出現する．

  例えば図 20(c) のコード片には，図 20(b) のソースコードでは出現しない変数 $var0$ が存在する．本手法ではコード片を生成するときに，型 t が必要な際にコンテキスト中に型 t に対応する変数がなければ新しく型 t の変数を宣言したり，全てのメソッド呼び

出しの戻り値を新しく宣言した変数へ代入している．そのため，開発者のソースコードでは出現する必要のない変数が生成したコード片上で出現する．

しかし，これらの違いは開発者がソースコードの編集を進める上で必要であれば変更が容易に可能であるため，変数の使用方法の違いよりも影響は小さいと考えられる．

6 関連研究

本研究に関連した先行研究として，開発者が編集しているソースコードを補完するためにコード片を推薦する手法や，既存ソフトウェアのソースコードを再利用可能なソフトウェア部品としてとらえ，開発者に対して有用なサンプルコードを提示する手法などが提案されている．本章ではこれらの本研究に関連した先行研究について述べる．

開発者が編集しているソースコードを補完するためのコード片を提示する手法としては [7, 11, 12, 14] などが提案されている．

Little らは，編集しているソースコードのコンテキストからメソッド呼び出しだけで構成される式を生成し，それらを開発者から与えられたキーワードにより順位付けを行い，順位の高いものを開発者へ提示する手法を提案した [11]．Mandelin らは，入力と出力となる 2 つの型が与えられると，入力の型の値から出力の型の値を生成するコード片を開発者へ提示する手法を提案した [12]．この手法では jungloid とよばれる頂点が型，辺がメソッド呼び出しである有向グラフを用いて変換を行うコード片を表現している．Sahavechaphan らは，既存のソースコード群と開発者が編集しているソースコードのコンテキストを利用して，指定された型の値を返すコード片を生成する手法を提案している [14]．この手法ではコンテキストから得られる利用可能な変数や，編集しているクラスの継承関係を利用してサンプルコードの絞り込みを行っている．Hill らは，イディオムのような有用で小さなコード片がコードクローンとして得られるとし，編集しているメソッドを完成させるために，そのメソッドと類似した構造をもつメソッドを探索する手法を提案した [7]．

本研究と，ソースコードを補完するためのコード片を提示する先行研究との主な違いは次の通りである．

- 制御構造要素を含むコード片を提示する．これにより開発者の行う処理だけでなく，それを行うための条件を開発者へ示すことができる．
- ソースコード上に出現するコーディングパターンを基にコード片を生成することで，特定の状況でしか利用できないようなコード片を除外することができる．
- 自然言語検索と同様に，単語に重みを与えることで開発者が重要でないキーワードを指定した場合でもそれに影響されない．

開発者にとって有用なサンプルとなるソースコードを提示する手法としては [8, 16, 17, 18] などの手法が提案されている．

島田らはソースコードから得た単語群をソースコードの特徴とし，潜在的意味インデキシング LSI により類似したソースコードを探索する手法を提案した [16]．渡邊らは 1 つのソースコードを 1 つの文書として捉え，文書に登場する単語の頻度から類似度を求める連想計

算を用いて編集中のソースコードに類似したソースコードを探索する手法を提案した [18]. Holmes らは編集中のソースコードに含まれるクラスの継承関係や出現する識別子などを利用して，ソースコード群から編集中のソースコードに類似したサンプルコードを探索する手法を提案した [8]．Ye らは開発者が行うソースコードの編集作業を監視し，開発者がメソッドの宣言を記述すると自動的に類似メソッドを探索する手法を提案した [17].

これらの先行研究は，既存のソースコード群から開発者が編集しているソースコードに類似したものを検索し，その検索結果を開発者へ再利用可能なソフトウェア部品として提示する．そのため，コーディングルールの具体的な使用例を開発者へ推薦するという本研究の目的とは適合しない．しかし，開発者が編集しているソースコードから自動的に収集した情報を用いてパターンの検索を行う場合には，これらの先行研究で提案されている手法が参考になると考えられる．

7　まとめと今後の課題

本研究では，ソフトウェアのパターン情報を基に，開発者が編集しているソースコードの補完を行うコード補完手法を提案した．具体的には，まず開発者が入力した目的の機能を表すキーワードに関連したパターンを検索する．そして，検索したパターンを基にソースコードの編集状況に適合したコード片を生成し，それを含むコード例を開発者へ自動的に推薦する．また提案手法をプログラミング言語 Java に適用したツールを Eclipse プラグインとして実装し，評価実験を行った．評価実験では，目的のパターンが重みの大きい単語をもつか，もしくは開発者が多くのキーワードを入力すれば，パターン集合から適切なパターンを取得することが可能であること，生成したコード片が多くの場合で妥当であることを確認した．

今後の課題としては，開発したツールの利便性の向上や，重みの小さな単語のみが出現するパターンを検索する際に，入力キーワードの数が小さい場合でも，目的のパターンが取得できるようなパターンの評価方法を考案することが挙げられる．パターンの評価方法を改善するためには，本手法におけるパターンの評価式を改良する以外に，パターンの検索の際にキーワード集合以外の情報を利用することが挙げられる．パターン検索の際に利用可能なキーワード集合以外の情報については次の通りである．

- キーワード列

  本手法では開発者から与えられたキーワードや，パターンから得る単語群を集合として扱った．すなわち，同一のキーワードが複数指定された場合，そのキーワードは一つしか指定されていないときと同様に扱い，またパターンに出現する単語は，その出現回数に関わらずパターンに 1 回しか出現しないものとした．そのため本手法では，単語の出現回数が異なるが同じ単語が出現するパターン間の違いを求めることが困難であった．

  Little らが提案した手法 [11] では複数のキーワードが指定されたとき，それぞれを独立したキーワードとして扱った．本手法でも同一のキーワードや単語をそれぞれ個別に扱うことで，図 21 のように同一の単語が複数のメソッド呼び出しで出現するようなパターンをより正確に指定することができると考えられる．

- 開発者が編集しているソースコードとパターンが出現するソースコードの関連性

  ソースコード群から得たパターン自身はその構成要素に関する情報しかもたないが，開発者が編集しているソースコードと，パターンが出現するソースコードとの関連性を求めることで，編集中のソースコードにより適したパターンを取得することができると考えられる．パターンが出現するソースコードから得られる情報としては，パターンの出現位置におけるクラスの継承関係やメソッドのコンテキスト情報，データフロー，

```
JEditBuffer.enableEditable()
JEDitBuffer.inputChar(char)
JEditBuffer.disableEditable()
```

図 21: 同一単語が複数回出現するパターンの例

制御フローなどが挙げられる．またこれらの情報は，パターンの評価方法の改善だけでなく，コード片の生成の際にも有用であると考えられる．

また本手法では，開発者が入力したキーワードがパターンに出現する単語に正確に一致するものとして扱っている．例えば開発者が目的とするパターンが図 21 のパターンであるとき，開発者は {buffer, editable, enable, disable} などの単語がキーワードであることを事前に把握していなければならない．しかし，実際には開発者が正確なキーワードを事前に把握することは困難だと考えられるため，開発者から与えられたキーワードに類似した単語を，自動的にキーワードとしてパターンの検索を行う方法が必要となる．

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石尾隆助教に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，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伊達浩典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] ASM. http://asm.ow2.org/.

[2] Eclipse. http://www.eclipse.org/.

[3] Fung: A Pattern Mining Tool for Java Method Calls. http://sel.ist.osaka-u.ac.jp/˜ishio/fung/.

[4] jEdit. http://www.jedit.org/.

[5] JHotDraw. http://www.jhotdraw.org/.

[6] PDF Render. https://pdf-renderer.dev.java.net/.

[7] Rosco Hill and Joe Rideout. Automatic Method Completion. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pp. 228–235, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 117–125, New York, NY, USA, 2005. ACM.

[9] Takashi Ishio, Hironori Date, Tatsuya Miyake, and Katsuro Inoue. Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs. In *WCRE '08: Proceedings of the 15th Working Conference on Reverse Engineering*, 2008.

[10] Donald E. Knuth. *The Art of Computer Programming, volume 3: (2nd ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[11] Greg Little and Robert C. Miller. Keyword programming in Java. *Automated Software Engg.*, Vol. 16, No. 1, pp. 37–71, 2009.

[12] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 48–61, New York, NY, USA, 2005. ACM.

[13] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. In *Proceedings of the 17th International Conference on Data Engineering*, pp. 215–224, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: mining For sample code. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 413–430, New York, NY, USA, 2006. ACM.

[15] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.*, Vol. 24, No. 5, pp. 513–523, 1988.

[16] Ryuji Shimada, Makoto Ichii, Yasuhiro Hayase, Makoto Matsushita, and Inoue Katsuro. Automatic software component recommendation using coding context. *IPSJ SIG Notes*, Vol. 2008, No. 112, pp. 31–38, 2008.

[17] Yunwen Ye and Gerhard Fischer. Reuse-Conducive Development Environments. *Automated Software Engg.*, Vol. 12, No. 2, pp. 199–235, 2005.

[18] 渡邉卓也, 増原英彦. 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. 日本ソフトウェア科学会第 24 回大会講演論文集, Vol. 24, pp. 3B–2, 2007.

[19] 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.