データマイニングを利用したプログラムの改善

田中 哲
akr@m17n.org

産業技術総合研究所 情報処理研究部門

概要

木構造に対するデータマイニングをプログラムの抽象構文木に適用すると、プログラム中に瀕出する構造を発見できる。このような構造はプログラムや言語自身を改善する候補となる。実際に Java および Ruby で記述されたいくつかのプログラムに対しデータマイニングを行ったところ、Java では Iterator による for ループ、Ruby ではクラスによる明示的なディスパッチなど、興味深い構造が発見できた。

1 はじめに

近年、XML を代表とする木構造のデータが蓄積されるようになり、それに伴って木構造に対するデータマイニングが研究されている [11, 16]。データマイニングとは、大量のデータから自明でない性質を発見するための技術である。木構造の性質とは、どのような構造が頻繁に現われるかということであり、データマイニングによって瀕出する構造を発見することが出来る。

プログラムコードは XML と同様な木構造であり、XML と同様にデータマイニングによって、瀕出する構造を発見できる。そのような構造を発見できるとすれば、様々な用途が考えられる。

- 特定のアプリケーション内で瀕出する構造が見つかった場合、これはリファクタリングの対象となり得る。したがって、リファクタリングでアプリケーションを整理できるかもしれない。
- あるライブラリを使うアプリケーション一般に瀕出する構造が見つかった場合、それらのアプリケーションをより単純に記述できるような API をライブラリに追加できるかもしれない。そのような API の追加はライブラリの改善となり得る。
- 複数のクラスの組み合わせの構造が見つかった場合、デザインパターンとしてまとめられる可能性がある。そう出来れば、ボキャブラリが増えたことになり、プログラムの構造に関するコミュニケーションを行いやすくなるかもしれない。
- ある言語で記述されたプログラム一般に瀕出する構造が見つかった場合、言語に追加する機能の候補の可能性がある。そのような追加によって、プログラムを単純化できるかもしれない。
- 特定のアプリケーション内で、特定のクラスのメソッドにはまとめられない構造が見つかった場合、アスペクトとしてまとめられるかもしれない。

一般に、プログラムに繰り返し出現する構造は重複したコード [13] としてプログラムが不適切なことを示しており、そのような構造をプログラマが手作業で繰り返し記述するのは生産性を落す原因になる。したがって、瀕出する構造をデータマイニングで発見し、何らかの抽象化機構でくくり出せれば、生産性を上げることが出来る。また、既存の抽象化機構でくくり出せない場合、新しい抽象化機構を検討する機会となる。

本稿では、実際にプログラムに対してデータマイニングを行い、見つかったパターンについて考察する。

2 木構造に対するデータマイニング

本稿で扱う木構造はラベル付順序木とする。木構造に対するデータマイニングとは、図 1 のように、入力として与えられるデータ木 (左) からパターン (右) を求めるものである。パターンは一般に複数存在し、図 1 の右にあげてあるのは一例である。

本稿におけるパターンは、ある木構造であり、ラベル・親子関係・兄弟関係を保存してパターンからデータ木の一部に対応づける写像が存在するもののことをいう。また、対応づけられたデータ木の部分をそのパターンの出現という。そして、パターンのルートに対応するデータ木のノードをルート出現という。

図 1 ではデータ木に番号をつけてあるが、パターンをラベル・親子関係・兄弟関係を保存してデータ木の 4,6,7 のノードに対応づけることが出来る。したがって、対応づけられた 4,6,7 はパターンの出現であり、パターンのルートに対応する 4 はルート出現である。また、4,6,7 の他には、1,2,11 と 1,8,11 と 8,9,10 という出現が

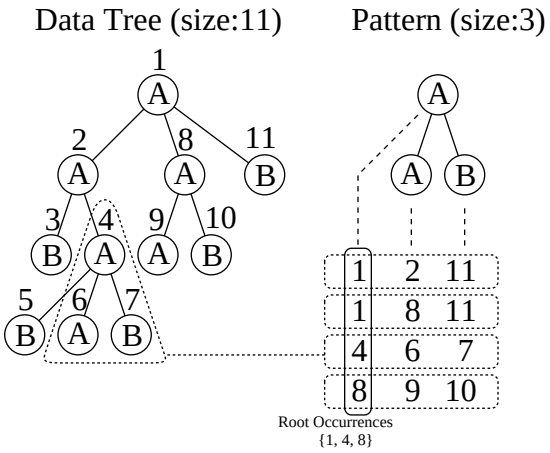

図 1: 木構造に対するデータマイニング

ある。それぞれのルート出現は 1,1,8 であり、4,6,7 のルート出現 4 も含めるとルート出現の集合は $\{1, 4, 8\}$ となる。

あるパターンが入力されたデータにどの程度現われるかをサポートといい、次式で表される。

$$\text{サポート} = \frac{\text{ルート出現の数}}{\text{データ木のノード数}}$$

ルート出現は 1,4,8 の 3 つであり、データ木のノード数は 11 なので、このパターンのサポートは $3/11 \fallingdotseq 0.27$ となる。

ここで、木構造に対するデータマイニングは次のように表現できる。

> データ木とサポートの下限を表す実数 σ が与えられたとき、サポートが σ 以上のパターンをすべて求める。$(0 < \sigma \leq 1)$

3 プログラムに対するデータマイニング

プログラムをデータマイニングする場合、プログラムを木構造として表現しなければならない。また、データマイニングの結果得られたパターンが、人間がすべて確認するには多すぎる場合、プログラムの改善に役立ちそうなパターンを優先して人間に提示していくことが必要になる。これらについて以下の節で述べる。

3.1 順序木によるプログラムの表現

プログラムの木による表現としては抽象構文木がある。これをデータマイニングするためには抽象構文木をラベル付順序木として解釈しなければならない。本稿では、抽象構文木の非終端記号・終端記号の名前をラベルとし、ノードの子の順序は対応するプログラムの記述順序と定義する。

なお、この順序づけはプログラムの意味をラベル付順序木として表現するという意味では近似であり、不適切な場合もある。たとえば、Java のクラス定義の並びの順序には意味がない。したがって、クラス定義の集合をデータマイニングする場合、得られるパターンはクラス定義が記述された順序に依存する。そして順序を無視すればひとつのパターンの出現とみなせる構造を発見できない可能性がある。

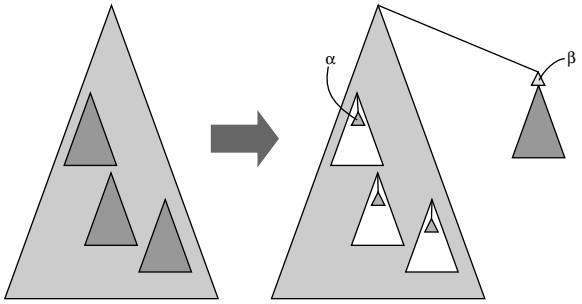

図 2: パターンの括り出し

3.2 改善候補となるパターンの抽出

プログラムに対するデータマイニングの目的はプログラムを改善に役立てる候補をユーザに提案することである。そのため、改善に役立ちそうな候補を優先してユーザに提示しなければならない。ここで、「改善に役立ちそうな」という基準をヒューリスティクスとして次式で表現する。なお、α, β は適当な定数である。

$$\text{パターンサイズ} \times \text{パターンの出現数} - \alpha \times \text{パターンの出現数} - \text{パターンサイズ} - \beta$$

この式は、図 2 のようにパターンの各出現をメソッドとしてくくり出せたと仮定した場合のプログラムサイズの減少を近似している。α はパターンの出現をメソッド呼び出しに置換したときのメソッド呼び出しのサイズを表現しており、括り出しにおいてはパターンの出現数だけプログラムは増加する。β はメソッドの定義からメソッドボディを除いたサイズを表現しており、括り出しにおいてはそのぶんだけプログラムは増加する。

また、興味のある対象が限定されている場合、パターンを限定して提示することが出来る。たとえば、プログラミング言語自身の改善に興味がない場合、ユーザプログラムの識別子 (メソッド名、変数名、クラス名

など) がパターンに含まれているものだけに限定できる。また、特定のライブラリの API にの改善に興味がある場合、その API に含まれるメソッドの呼び出しがパターンに含まれているものだけに限定できる。

4 データマイニング実験

木構造に対するデータマイニングアルゴリズム FREQT[11] を Java で実装[1]し、Java および Ruby で記述されたいくつかのプログラムに対して適用した。最小サポートは 2/ノード数 とし、複数の出現が存在すれば検出できるように設定した。また、処理がメモリ内で収まるように、パターンのサイズは 20 を上限とし、さらに探索の途中で横型探索のキューが 1000 以上になったら探索を打ち切ることとした。

Java プログラムの構文解析は JavaCC を利用して記述した構文解析器を使用した。Ruby プログラムの構文解析は btyacc を利用して記述した構文解析器を使用した。

また、検出されたパターンは 3.2 節で述べた基準にしたがって並べて目視で調査したが、その際、α, β は 5 とした。

4.1 Apache Ant

Apache Ant 1.6 に対してデータマイニングを行った。Ant[2] は Java で記述されたビルドツールで、ソースは約 8MBytes、93 万行である。構文解析を行って木構造に変換すると 495621 ノードとなった。

この結果、37889 パターンが報告された。それらを 3.2 節で述べた基準にしたがって並べ、目視で調査した。その中で発見された中で、興味深いものをいくつか述べる。

まず、発見された中でもっともプログラム改善に役立ちそうなものとされたのは次のようなメソッドの定義部分である。

TYPE *METHODNAME*(*TYPE* *VAR*)

このパターンは構文木では図 3 のように表現されており、サイズは 8 である。これが 4098 回出現した。しかし、このパターンはアプリケーションやライブラリなどに依存しない当り前の構造であり、プログラムの改善には役立たない。

また、識別子が含まれるパターンの中でもっとも役立ちそうなものとして、次のパターンが見つかった[2]。

[1]正確には FREQT そのものではなく、冗長なパターンをなるべく提示しない工夫をいくつか行っている。

[2]??? として不明な部分を示してあるが、この場所はパターンからは判明しない。一般にはほかの位置にもパターンには現われていないノードが存在する可能性がある。

```
MethodDeclaration
  ├ResultType
  └MethodDeclarator
    └FormalParameters
      └FormalParameter
        ├Type
        │└Name
        └VariableDeclaratorId
```

図 3: メソッド定義パターンの構文木

```
CompilationUnit
  ├PackageDeclaration
  │└Name
  ├ImportDeclaration
  │└Name
  │  ├ID-org
  │  ├ID-apache
  │  ├ID-tools
  │  └ID-ant
  └ImportDeclaration
    └Name
      ├ID-org
      ├ID-apache
      ├ID-tools
      └ID-ant
```

図 4: import パターンの構文木

```
package NAME;
import org.apache.tools.ant.???;
import org.apache.tools.ant.???;
```

構文木は図 4、サイズは 15 で、465 回出現している。これはファイル先頭の import 宣言を示している。また、`org.apache.tools.ant` という記述が 2 回現われていることから、Java に何らかの短縮記法を導入して 1 回で済ませるようにすれば、記述を単純化することが可能である。つまり、データマイニングのこの結果は Java の改善可能な箇所を提示したことになる。ただし、そのような記法の導入は言語自身の単純さを損なうことになるため、必ずしも言語仕様全体として適切だとは限らない。

また、77 回出現する次のパターンも見つかった。こ

```
ForStatement
  ├ForInit
  │└LocalVariableDeclaration
  │  ├Type
  │  │└Name
  │  │  └ID-Enumeration
  │  └VariableDeclarator
  │    └VariableDeclaratorId
  ├Member-hasMoreElements
  │├PrimaryBase
  │└Arguments
  └LocalVariableDeclaration
    ├Type
    │└Name
    └VariableDeclarator
      ├VariableDeclaratorId
      └CastExpression
        ├Type
        │└Name
        └Member-nextElement
```

図 5: Iterator によるループパターン

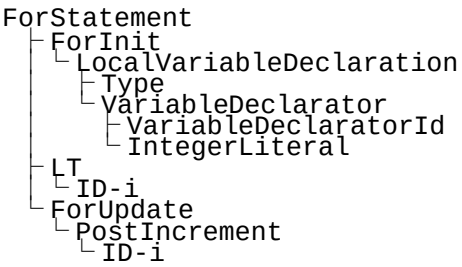

図 6: 整数のインクリメントによるループ

のパターンは構文木としては図 5 であり、ノード数は 20 である[3]。

```
for (Iterator VAR = ...; VAR.hasNext();) {
  TYPE VAR = (TYPE) VAR.next();
  ...
}
```

このパターンは Iterator による繰り返しコードを示している。このパターンは Ant に限らず Java プログラム一般に瀕出し、実際他の Java プログラムにも見つかった。つまり、このようなパターンが瀕出するのは Java という言語の性質であり、これを簡潔に書けるような仕組みを用意できれば、生産性を向上できる。

これを簡潔に記述するには、例えばこのループを略記する構文を言語に導入すればよい。実際、J2SE 1.5 (Tiger) ではこれを略記するための記法が導入される [5, 1]。つまり、データマイニングによって Java の問題点を検出できたことになる。

4.2 Apache Tomcat

Apache Tomcat 5.0.16 に対してデータマイニングを行った。Tomcat[7] は Java で記述されたサーブレットコンテナで、ソースは約 12MBytes、134 万行である。構文解析を行って木構造に変換すると 726924 ノードとなった。

データマイニングの結果、45798 パターンが見つかった。以下では興味深いものをいくつか述べる。

識別子が含まれたパターンのうち 2 番目のものとして、次のパターンが 782 回出現しているのが見つかった。これは構文木では図 6 の構造であり、ノード数は 12 である。

```
for (TYPE VAR = INT; i < EXP; i++)
```

このパターンは整数のインクリメントによるループを示している。

[3] 20 というのはパターンサイズの上限として設定した値であるため、上限を大きくすればこのパターンを含むより大きなパターンが発見される可能性もある。

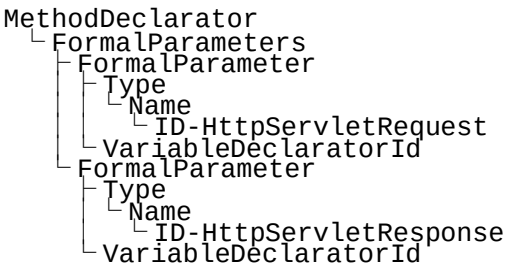

図 7: HttpServletRequest, HttpServletResponse を仮引数にもつメソッド定義

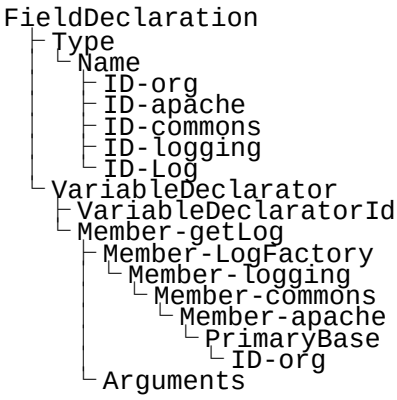

図 8: LogFactory.getLog によるフィールド宣言

また、`i` や `String` などの普遍的な識別子ではない、アプリケーション依存な識別子が出現するパターンとしては、263 回出現した次のパターンが見つかった。

```
TYPE METHODNAME(HttpServletRequest VAR,
                HttpServletResponse VAR)
```

パターンの構文木は図 7 で、ノード数 12 である。このパターンは、仮引数として HttpServletRequest と HttpServletResponse を持つメソッドが多数現われていることを示している。

また、次のパターンが 58 回出現した。

```
org.apache.commons.logging.Log VAR =
  org.apache.commons.logging.LogFactory.getLog(???);
```

パターンの構文木は図 8 で、ノード数 18 である。このパターンは Jakarta Commons の Logging を使う場合に各クラスに必要となるフィールド宣言 [3] を示している。

このような定型的なフィールド宣言は AspectJ のような言語を使用すれば、ひとつのアスペクトにまとめて記述できる可能性がある。

4.3 Xerces, JDOM, XOM のサンプル

3 種類の Java 用 XML 処理ライブラリ (Xerces[9], JDOM[4], XOM[10]) に添付されているサンプルプロ

グラムに対してデータマイニングを行った。Xerces は W3C で標準化された DOM[8] という API の実装であり、JDOM と XOM は DOM よりも使いやすい API を目指して設計されたライブラリである。

Xerces-2.6.0 Xerces のサンプルプログラムは約 680KBytes、6 万行である。構文解析を行って木構造に変換すると 57753 ノードとなった。データマイニングを行うと 13017 パターンが検出された。

データマイニングで得られたパターンのうち、Xerces が提供するメソッドの呼び出しを含むものでもっとも改善が可能らしいものは 78 番目に現われる次のパターンであり、setFeature というメソッド呼び出しを含んでいる。これは 36 回出現してノード数は 17 である。

```
TryStatement
├─Member-setFeature
│ ├─PrimaryBase
│ └─Arguments
├─FormalParameter
│ ├─Type
│ │ └─Name
│ └─VariableDeclaratorId
└─Member-println
  ├─Member-err
  │ └─PrimaryBase
  │   └─ID-System
  └─Arguments
    └─Plus
      ├─Plus
      │ └─StringLiteral
      └─StringLiteral
```

JDOM-b9 JDOM のサンプルプログラムは約 170KBytes、2 万行である。構文解析を行って木構造に変換すると 7578 ノードとなった。データマイニングを行うと、10625 パターンが検出された。

データマイニングで得られたパターンのうち、JDOM が提供するメソッドの呼び出しを含むものでもっとも改善が可能そうなものは、930 番目に現われる次のパターンであり、build というメソッド呼び出しを含んでいる。これは 6 回出現してノード数は 20 である。

```
UnmodifiedClassDeclaration
└─ClassBody
  └─MethodDeclaration
    ├─ResultType
    ├─MethodDeclarator
    │ └─FormalParameters
    │   └─FormalParameter
    │     ├─Type
    │     │ └─Name
    │     │   └─ID-String
    │     └─VariableDeclaratorId
    ├─Throws
    │ └─NameList
    │   └─Name
    └─LocalVariableDeclaration
      ├─Type
      │ └─Name
      │   └─ID-Document
      └─VariableDeclarator
        └─Member-build
```

XOM-1.0d22 XOM のサンプルプログラムは約 310KBytes、4 万行である。構文解析を行って木構造に変換すると 28047 ノードとなった。データマイニングを行うと、9945 パターンが検出された。

データマイニングで得られたパターンのうち、XOM が提供するメソッドの呼び出しを含むものでもっとも改善が可能そうなものは 347 番目に現われる次のパターンであり、build というメソッド呼び出しを含んでいる。これは 44 回出現してノード数は 20 である。

```
MethodDeclaration
├─ResultType
├─MethodDeclarator
│ └─FormalParameters
│   └─FormalParameter
│     ├─Type
│     │ └─Name
│     └─VariableDeclaratorId
└─TryStatement
  ├─LocalVariableDeclaration
  │ ├─Type
  │ │ └─Name
  │ └─VariableDeclarator
  │   ├─VariableDeclaratorId
  │   └─Member-build
  │     ├─PrimaryBase
  │     └─Arguments
  └─FormalParameter
    └─Type
      └─Name
```

3 種類のデータマイニングで、各ライブラリの API で提供されるメソッド呼び出しが現われる順位を比べると、Xerces(78 番目), XOM(347 番目), JDOM(930 番目) の順になる。このことは、Xerces を使うプログラムでは、定型的なコードをもっとも多く書かなければならないことを示している。つまり Xerces(DOM) は JDOM や XOM よりも冗長な記述を要求する API であると判断でき、これは DOM よりも使いやすいものを求めて JDOM や XOM が作られたという経緯に一致する。つまり、データマイニングによってライブラリ API の善し悪しを判断できた可能性がある。

4.4 Ruby 標準添付ライブラリ

Ruby 1.8.1 に標準添付されているライブラリに対してデータマイニングを行った。Ruby[6] はスクリプト言語であり、スクリプトから使用するための様々なライブラリが添付されている。それらのライブラリのうち、Ruby 言語で記述されているものは約 2.4MBytes、27 万行である。これを構文解析を行って木構造に変換すると 233185 ノードとなった。

データマイニングによって 34558 パターンが発見された。

発見されたパターンのうち、問題点を提示していると考えられるものとしては、次のものがある。このパターンは 213 回出現し、構文木は図 9 で、ノード数 6 である。

```
cond
  └m_kind_of?
    ├recv
    │ └variable
    └arg1
      └variable
```

図 9: Ruby における kind_of? の呼び出し

```
if VAR.kind_of? VAR
```

このパターンは if 文や case 文などの条件節 (cond) で kind_of? というメソッド呼び出しを行っていることを示している[4]。kind_of? はレシーバが引数に与えたクラスに属するかどうかを判定するメソッド[5]であるが、Ruby ではオブジェクトの種類の判別にはクラスでなくメソッドの存在で行うという方針があり、クラスで判別するのは悪いスタイルである[6]。しかし、kind_of? を含むパターンが出て来ることは、その方針に従っていないライブラリが多数あることを示している。

4.5 prettyprint.rb の改善

また、Ruby に添付されたライブラリの中で、prettyprint.rb というライブラリの API の改善に興味があるとして、prettyprint.rb 中の PrettyPrint クラスで定義される first? というメソッドの呼び出しを含むものを抽出した。ここで、first? の呼び出し自体は全部で 5ヶ所あり、そこから 12 種類のパターンが抽出された。この 5ヶ所の呼び出しはすべて pp.rb というファイル中にあり、抽出された 12 種類のパターンの出現もすべて pp.rb 内にある。

ここで、prettyprint.rb は、Wadler の pretty printing 代数 [15] の実装である。また、pp.rb は Ruby のオブジェクトの内容を複数行に渡って清書するライブラリであり、prettyprint.rb を利用して改行とインデントの制御を行っている。

これらのパターンの中に、次のパターンが見つかった。このパターンは 3 回出現し、構文木は図 10 で、ノード数 11 である。

```
EXP.each {
  unless first?
    ...
  end
  group {
    pp ...
  }
}
```

[4]Ruby コードで示したパターンは if 文の場合である。

[5]is-a 関係を判定するメソッド

[6][ruby-list:39022] http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-list/39022

```
m_each
  ├recv
  └block
    ├unless
    │ ├cond
    │ │ └m_first?
    │ └then
    └m_group
      └block
        └m_pp
          └arg1
```

図 10: first? に関わるパターン

ここで、外側の each は Ruby ではループを実現する標準的なメソッド名である。また、first? は最初に呼び出されたときだけ真を返し、それ以降の呼び出しでは偽を返すメソッドである。つまり、このパターンはループの最初以外の繰り返しにおいて何らかの処理を行う記述を示している。実際にパターンの出現を確認すると、リストの各要素の間にカンマを挿入して表示するという記述であった。

ここで区切りを挿入しつつ繰り返しを行うメソッドとして、図 11 の seplist を PrettyPrint に追加し、seplist を使用して first? が呼び出されている周辺を書き換えたところ、first? が使用されている 5ヶ所すべてを seplist を使用して書き換えることが出来た。そのうちの一ヶ所を図 12 に示す。その各出現毎にコードは 1 行減り、計 5 行減少した。また、first? を使用していないものの、seplist を使用できる部分が set.rb に 1ヶ所存在し、これを書き換えたところ、図 13 のように 7 行減少した。

つまり、seplist の 10 行の追加に対し、合計で 12 行の減少を実現できた。これはノード数では 30 ノード増、28 ノード減となる。したがって、prettyprint.rb, pp.rb, set.rb の範囲ではコードの増加と減少はほぼ同量である。しかし、first? は公開 API であるため、Ruby に添付されていないプログラムでも使用されている可能性がある。そのようなプログラムでも同様に seplist の使用によるコード減少の利点が得られると考えれば、Ruby に添付されているライブラリ自体のコード量が減らないといっても、ライブラリとして seplist を提供する価値はある。

また、これらの書き換えにより first? は使用されなくなったことにより、first? の定義を削除できる[7]。これを行うと 22 行減少した。これは 45 ノードの減少となる。

つまり、最終的には表 1 のように 24 行、43 ノードの減少が可能である。また、他のプログラムでも区切りを挿入しつつ繰り返しを行うコードがあれば、さらに多くのコード減少が見込める。

[7]ただし、first? は公開 API であるため、実際に削除する前には周知するための期間をとらなければならない。

表 1: seplist によるコードの増減

ファイル	`seplist` の追加	`seplist` の使用	`first?` の削除	合計
prettyprint.rb	+10line +30node		−22line −45node	−24line −43node
pp.rb, set.rb		−12line −28node		

```
def seplist(list, sep=nil, iter_method=:each)
  sep ||= lambda { comma_breakable }
  first = true
  list.__send__(iter_method) {|*v|
    if first
      first = false
    else
      sep.call
    end
    yield *v
  }
end
```

図 11: seplist の実装

```
obj.each {|k, v|
  comma_breakable unless first?
  group {
    pp k
    text '=>'
    group(1) {
      breakable ''
      pp v
    }
  }
}
```

→

```
seplist(obj, nil, :each_pair) {|k, v|
  group {
    pp k
    text '=>'
    group(1) {
      breakable ''
      pp v
    }
  }
}
```

図 12: pp.rb 中の seplist への書き換え

```
pp.nest(1) {
  first = true
  each { |o|
    if first
      first = false
    else
      pp.text ","
      pp.breakable
    end
    pp.pp o
  }
}
```

→

```
pp.nest(1) {
  pp.seplist(self) { |o|
    pp.pp o
  }
}
```

図 13: set.rb 中の seplist への書き換え

5 関連研究

プログラム内部からコードの複製やリファクタリング候補の発見を行う研究にはさまざまなものがあり [14]、プログラムを構文木に変換して処理を行うものには [12] がある。しかし、[12] で使われているアルゴリズムはアドホックで、本稿で用いた FREQT というデータマイニングアルゴリズムよりも見通しが悪い。ただし、アルゴリズム内ではプログラム言語に特化した工夫が行われており、そのような工夫を FREQT にうまく導入することは今後の課題である。

6 まとめ

一般に、プログラム中に繰り返し現われる記述は重複したコードとしてプログラムが不適切なことを示しており、一ヶ所にまとめて抽象化する候補と考えられる。したがって、データマイニングによって繰り返し出現するパターンが発見できれば、それは抽象化する候補になる。たとえば、発見された候補をリファクタリングしてメソッド抽出できれば、プログラムを改善できる可能性がある。そして、抽出したメソッドをライブラリで提供できれば、ライブラリの API を使いやすく改善することになる。また、ライブラリに抽出できない場合でも、言語機構に出来る場合もある。

本稿ではいくつかのプログラムを対象にデータマイニングを行い、発見された次のパターンについて述べた。

- 同じような import の並び
- Iterator を使用した for ループ
- 整数のインクリメントによる for ループ
- servlet のメソッド定義
- Jakarta Commons の Logging での各クラスの定型的なコード
- Ruby での条件節における kind_of? の使用
- 区切りを挿入する繰り返し

これらのパターンはプログラムや言語の改善に利用できる可能性があり、実際、Iterator によるループなど改善が行われているものもある。

また、3 種類の XML パーザのサンプルプログラムに対してデータマイニングを行い、ライブラリ API の良否を評価することを簡単に試みた。その結果、DOM(Xerces) よりも良いものを求めて JDOM や XOM が作られたという経緯に沿った結果が得られた。

なお、当初期待していた複数のクラスにかかわるデザインパターンは発見できなかった。これは、今回使用した FREQT が順序木を対象としているため、クラス間に順序を導入しなければならず、また、名前による参照で構成された構造を発見できないためである。デザインパターンを発見するためには、おそらくグラフ構造を対象としてデータマイニングを行う必要がある。

参考文献

[1] An enhanced `for` loop for the Java Programming Language. `http://jcp.org/aboutJava/communityprocess/jsr/tiger/enhanced-for.html`.

[2] Ant home page. `http://ant.apache.org/`.

[3] Jakarta Commons Logging Users Guide. `http://jakarta.apache.org/commons/logging/userguide.html`.

[4] JDOM home page. `http://www.jdom.org/`.

[5] JSR 201: Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import. `http://www.jcp.org/en/jsr/detail?id=201`.

[6] Ruby home page. `http://www.ruby-lang.org`.

[7] Tomcat home page. `http://jakarta.apache.org/tomcat/`.

[8] W3C Document Object Model. `http://www.w3.org/DOM/`.

[9] Xerces home page. `http://xml.apache.org/xerces2-j/index.html`.

[10] XOM home page. `http://cafeconleche.org/XOM/`.

[11] 浅井達哉, 安部賢治, 川副真治, 坂本比呂志, 有村博紀, 有川節夫. 半構造データからの頻出パターン発見アルゴリズム. 第１３回データ工学ワークショップ（DEWS2002）, March 2002.

[12] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM'98 (International Conference on Software Maintenance)*, pp. 368–377, 1998.

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

[14] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques. In Michael W. Godfrey Tom Mens, Juan F. Ramil and Brian Down, editors, *Proceedings ELISA'03 (International Workshop on Evolution of Large-scale Industrial Software Applications)*, pp. 25–36. Vrije Universiteit Brussel, September 2003.

[15] Philip Wadler. A prettier printer, March 1998. `http://homepages.inf.ed.ac.uk/wadler/topics/language-design.html#pretti%er`.

[16] M. Zaki. Efficiently mining frequent trees in a forest. ACM, July 2002. In Proc. SIGKDD 2002.