シーケンスの構造派生機能を有する検索モデルのソースコード検索への応用

宇田川　佳久*

Derived Sequence Retrieval Model and its Application to Source Code Retrieval

Yoshihisa Udagawa*

In this paper, an approach that improves source code retrieval using structural information of source code is presented. I propose a similarity measure that is defined by the ratio of the number of sequential fully matching statements to the number of sequential partially matching statements. This paper discusses the relationship among the proposed similarity, the Sorensen-Dice index and the cosine similarity. Retrieval experiments are performed using Struts 2, which is a medium-size open source Java program. The results show that the proposed similarity is a better indicator than the others.

はじめに

情報システムが多くの社会・経済活動で使われるようになった昨今、信頼性と可用性に優れた情報システムが求められている。高品質なソフトウェアを開発・管理する基本は、ソースコードを確認することであるが、実用的なシステムは数万行から数十万行に及ぶソースコードで構成されており[1]、十分な確認が実施されていないのが現状である。

一般に、ソフトウェア開発では、数行の実行文で構成されるコードをコピーしてプログラムを作成することが行われており、類似したソースコード（クローン）がソースコードに散在する。Baker 氏[2]および Mayrand 氏ら[3]は、典型的な実務向けのソフトウェアの 7% から 23%のソースコードがクローンであることを論じている。

これまでに、クローン検出、流用箇所検出およびソフトウェア保守を目的とした多数のソースコード検索手法が提案されている。これらは下記の 4 種類に大別できる。

(1) テキスト比較

この手法は、ソースコードを文字列として捉え、ソースコードに繰返し発生する文字列を抽出するものである。Baker 氏[2] は、パラメータ化された文字列（p-string）と 2 つの p-string 間の一致（p-match）を定義し、p-match を高速に実行するための接尾辞木（p-suffix tree）を提案している。Marcus 氏らは[4] 情報検索の分野で開発された潜在的意味インデキシング（Latent Semantic Indexing）を使った検索方法を提案している。テキスト比較による方法は、文字列を基本として検索するため、変数名の書き換えなどの影響を受けやすい。

(2) トークン比較

この方法は、ソースコードに含まれる識別名や計算式を一定のルールに基づいて生成される識別名（トークン）に置き換えた後に、ソースコードを比較するものである。肥後氏ら[5] および神谷氏ら[6] は、プログラミング言語の文法に則って変数や定数をトークン列に変換し、このトークン列に於いて類似したソースコードを検索する手法を論じている。トークンの置替えルールをソースコードの構文解析の段階で指定することができ、複数のプログラミング言語への適用が可能である。構文解析では、制御文などのプログラム構造はトークンに置き替えないので、プログラム構造を反映したソースコードの類似検索が可能である。CodeDepot[7] は、単純な文字列検索機能やトークンを使った類似ソースコードの検索機能などを提供するシステムであり、実務における大規模ソースコードの処理を対象としている。CP-Miner [8] は、閉じた頻出パターンのマイニング（Frequent Closed Pattern Mining）手法を使って、トークン化されたソースコードから頻出するソースコード片を検出するものである。一般に、トークン比較による方法は、テキスト比較による方法よりも変数名や計算式の書き換えの影響を受けにくく、大規模なソースコードに対して適用可能である。

(3) 構造比較

ソースコードを構文解析することによって得られる抽象構文木（Abstract Syntax Tree）や、プログラム依存グラフ（Program Dependency Graph）を使って比較を行う方法である。CloneDR [9] は、抽象構文木を使ったクローン検出の先駆的な研究である。DECKARD [10] は、構文解析木の一部分をベクトルによって近似し、ユークリッド距離に基づく類似度を使って類似する構文解析木をクラスタリングする方法を提案している。なお、構文解析木をベクトルで近似する程度については、検索アルゴリズムのパラメータとして指定することができる。

プログラム依存グラフは、制御フローの情報とデータフローの情報から構成される。プログラム依存グラフに対し

* 東京工芸大学工学部コンピュータ応用学科教授
2014 年 9 月 29 日 受理

て、部分グラフ同型一致（Isomorphic Subgraph Matching）アルゴリズムを適用することで、クローンを検出する。Komondoor 氏ら[11] は、C プログラムのクローンを検出する手法を論じている。Krinke 氏[12] は、最大一致部分グラフを効率的に検出するために、パスの長さ k（任意の正数）をパラメータとする手法（k-limited Path Matching)を提案し、再現率、精度、性能について論じている。一般に、構造比較を使ったクローンの検出は、計算コストが高いことが知られている。

(4) ソフトウェア指標比較

ソースコードを比較・分析するための様々なソフトウェア指標が提案されている。例えば、循環的複雑度（Cyclomatic）[13]、オブジェクト指向における継承の深さ(DIT: Depth of Inheritance Tree)やサブクラスの数(NOC: Number Of Children) [14]などがある。ソフトウェア指標の類似性の高いものをクローンとして検出する研究がある[15][16]。一般に、ソフトウェア指標を使ったクローンの検出は、ファイル（クラス）あるいはメソッドといったソースコードの大きな単位での比較には効果があるが、特定のソースコードの断片を特定することが難しい。

本研究は、構造比較に分類されるものであるが、従来の手法とは、下記の点において異なる。

- 前処理において Java ソースコードに現れる変数の型を Java の文法に準拠した静的解析を実施し、その結果に基づいて変数名を変数の型に置き替える。これにより、変数の型を正確に反映した検索ができる。
- 検索処理においては、検索条件として与えられたシーケンスから部分シーケンスを派生させて検索を実行する、独自の検索方式を使用している。この検索方式をシーケンス構造派生検索モデル（DSRM: Derived Sequence Retrieval Model）と呼ぶ[17]。

本文で提唱するシーケンス構造派生検索モデルは、2 個の集合の類似度を計算する指標である Sorensen-Dise 係数（Jaccard 係数の変形）の拡張として定義される。また、一般的な文書の検索手法として広く用いられているベクトル空間モデルは、集合を構成する要素に重み付けをしたベクトルに基づいて集合間の類似度を計算する指標であり、Sorensen-Dise 係数の一般形と見なすことができる。本文では、シーケンス構造派生検索モデル、拡張 Sorensen-Dise 係数、ベクトル空間モデルの関連性を論ずるとともに、Java ソースコードを対象とした評価実験によって、構造派生検索モデルが、他の検索手法よりも高い絞り込みを達成していることを示す。

以降、研究のプロセスと本研究で開発した主な検索機能、実験の対象とした Struts 2 の Java ソース、ソースコードの類似指標、検索実験と評価の順に述べる。

研究のプロセス

開発した主な検索機能

図 1 は、開発したツールの構成を示している。構造抽出ツールは、Java ソースコードを解析し、メソッドごとに、制御文と呼び出しているメソッド名をネスト構造とともに抽出する。構造抽出ツールは、処理速度と移植性を考慮し ANSI 規格準拠の C 言語で実装されている。

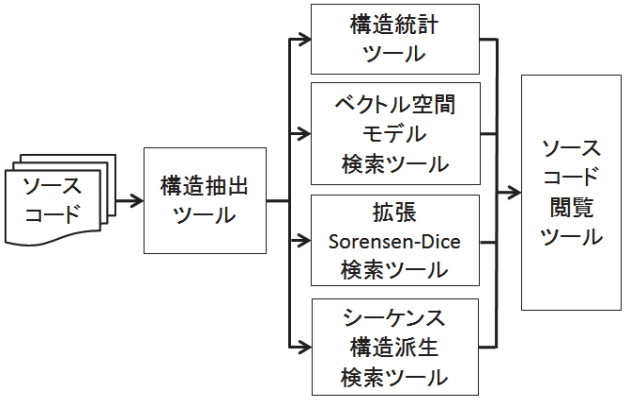

図 1　開発したツールの構成

「構造統計ツール」は、制御文やメソッド名の発生個数、ネストの最大深さ、最大行数などの統計情報を算出する。「ベクトル空間モデル検索ツール」は、制御文やメソッド名の発生個数をベクトル空間モデルの“文書ベクトル”の要素値に対応させ、類似度を算出するツールである。「拡張 Sorensen-Dice 検索ツール」は、Sorensen-Dice 係数を重複する要素を含む集合（多重集合）に拡張したモデルに基づいた類似検索を行うツールである。「シーケンス構造派生検索ツール」は、制御文およびメソッド名で構成された文のシーケンス（並び）をキーにした検索を行うツールである。この検索の特徴は、指定されたシーケンスに n 個の要素が含まれるとき、2n − 2 個の派生シーケンスを生成し、これらのすべてのシーケンスを使って検索を実行することである。「ソースコード閲覧ツール」としては商用のソースコード解析ツールを採用している。

構造抽出ツールの主な機能

構造抽出ツールが抽出対象とする要素は下記の通り。

(1) クラス名、メソッド名と引数

「構造抽出ツール」は、クラス内に定義されたメソッドごとに、制御文とメソッドの呼び出しを抽出する。通常、Java のクラスは、[public | protected | private] class **クラス名{変数定義|メソッド定義}** という構文で宣言される。クラス名、メソッド名と引数は、メソッドが定義されているクラス名とともに、下記の構文で抽出する。

クラス名::メソッド名(引数)

Java の匿名クラス（Anonymous Class）[18]は無名クラスとも呼ばれているものであり、構文的には、クラス宣言を持たず、クラスのインスタンス化を行う new クラス名() に続いて実装で必要な変数定義およびメソッド定義を行う

内部クラスの一種である。匿名クラスは、明示的なコンストラクタを持てない、継承するクラスを作成できないと言った制約があるが、処理が単純で、別途再利用する予定もない抽象クラスおよびインタフェースクラスを実装するときに使われることが多い。構造抽出ツールは、匿名クラスが定義されている場合は、下記の構文で抽出する。

クラス名:匿名クラス名:メソッド名(引数)

構造抽出ツールは、総称型(Generics)の型パラメータをメソッドの構文情報として抽出する。総称型に関する情報は、メソッドの同一性の判定に使用され、例えば、HashMap<Integer, String> と HashMap<String, Integer> は別物として処理される。

(2) Java の制御文

構造抽出ツールは、Java の制御文[18] をネスト構造とともに抽出する。ブロック構造は、"{" と "}" で表現する。従って、"{"の数がネストの深さを表す。Java で定義されている以下の制御構造を抽出対象とする。

- *if* 文（*else*、*else if* のバリエーションを含む）
- *try* 文（*catch*、*finally* のバリエーションを含む）
- *switch* 文
- *while* 文
- *do while* 文
- *for* 文
- *break* 文
- *continue* 文
- *return* 文
- *throw* 文
- *synchronized* 文

(3) メソッドの中で呼び出しているメソッド名

構造抽出ツールは、メソッドの中で呼び出しているメソッド名を抽出対象とする。また、変数名とその変数が値域とするクラス名またはデータ型の対応関係を抽出しており、**変数名.メソッド名**が出現した場合は、変数名を該当する **クラス名.メソッド名** または **データ型.メソッド名** に置き替える処理を行っている。これにより、変数名の違いによる影響を取り除いている。ただし、変数の型の違いは検索結果に反映される。例えば、**変数名１.add**(<String>) と **変数名２.add**(<String>)とがあり、それぞれ、LinkedList <String>.add(<String>) と List <String>.add(<String>) に変換されたとき、これらは別のメソッドとして扱われる。

Struts 2 からの構造抽出

Struts および Struts 2 は、Apache Software Foundation[19] が無償で提供している Java Web アプリケーション・フレームワークである。Struts は、2000 年前半に、実務向け Web システムの開発でも広く利用された実績がある。Struts 2 は、Struts のアップデート版として 2007 年に公開された。Struts 2 の特徴として、ActionForm 関連の記述がなくなりソースコードの量が軽減できる、アプリケーションクラスの実行の前後処理が充実しているなどの特徴があり、今後の普及が期待されている[20]。

本研究で対象としたのは Struts 2.3.1.1 の Core と呼ばれているソースプログラムであり、Struts 2 が提供する主要な機能に対応した *components*、*config*、*dispacher*、*intercepter*、*views* などのパッケージから構成されている。Struts 2.3.1.1 Core ソースコードに関する主なメトリックスは下記の通り。

ファイル数:	368 個
クラス数:	414 個
メソッド数:	2,667 個
宣言文と実行文の行数:	21,543 行
コメント行数:	17,954 行
ソース総行数:	46,100 行 (空白行を含む)
最大ネスト数:	8
最大循環的複雑度:	55

Struts 2.3.1.1 Core のソースコードの規模は、中規模システムと位置付けられるものであり[1]、本研究で開発したツールの機能と性能を確認する上で適切な規模であると考えられる。

図 2 は、*org.apache.struts2.views.velocity.components* パッケージ内の *AbstractDirective* クラスの *putProperty()*メソッドについて、制御文とメソッド名の構造を抽出した結果である。#記号に続く 3 個の数字は、それぞれ、コメント行数、空白行数、コード行数を示している。構造抽出ツールはネスト構造の情報も抽出するので、制御文とメソッド名の部分構造によるマッチングに必要な情報を提供している。図 3 は、図 2 の抽出元になった *putProperty()*メソッドのソースコードである。

```
AbstractDirective::void putProperty()
#     4    0  11
{
node.value
String.indexOf
    if{
    String.substring
    String.substring
    propertyMap.put
    }
    else{
    throw new ParseErrorException
    }
}
```

図 2　構造抽出結果の例

```
protected void putProperty(Map propertyMap,
  InternalContextAdapter contextAdapter, Node node)
  throws ParseErrorException, MethodInvocationException {
  // node.value uses the StrutsValueStack ...
    String param = node.value(contextAdapter).toString();
    int idx = param.indexOf("=");
    if (idx != -1) {
      String property = param.substring(0, idx);
      String value = param.substring(idx + 1);
      propertyMap.put(property, value);
    } else {
      throw new ParseErrorException
        ("#" + this.getName() + " is illegal!");
    }
  }
```

図 3　*putProperty()*メソッドのソースコード

ソースコードの類似指標

ベクトル空間モデル

ベクトル空間モデルは、文書中の単語の発生数に基づいて文書をベクトルとして表現し、文書同士をベクトル空間上で比較することによって文書間の類似度を計算する数理モデルのひとつである。文書に含まれる単語をベクトルの要素（次元）に割り当てることにより、一つの文書を一つのベクトルとして扱う。

D を検索対象とする文書集合、$\boldsymbol{d_j}$ を D に属する文書とするとき、$\boldsymbol{d_j}$ を次のベクトルで表現する。

$$\boldsymbol{d_j} = (w_{1,j}, w_{2,j}, \ldots, w_{N,j})$$

ここで N は D に含まれる単語の数、$w_{i,j}$ は i 番目の単語の重要度（重み）である。ベクトル要素 $w_{i,j}$ は、単純に単語の出現頻度とすることも可能であるが、*tf-idf* 法と呼ばれている重み付けが広く用いられている[21]。文書 j における単語 i の出現頻度を $tf_{i,j}$ (Term Frequency)、単語が出現する文書の割合を idf_i (Inverse Document Frequency) と呼び、両者の積をその文書における単語の重要度とする。

$$w_{i,j} = tf_{i,j} \cdot idf_i$$

idf_i は、全文書数を m、単語 i を含む文書数を df_i とするとき

$$idf_i = log_2\left[\frac{m}{df_i}\right]$$

で計算する。idf_i は、単語 i が全文書において発生する頻度が低いほど大きな値になる。すなわち、発生頻度が低い単語ほど、重要度が高い有効語として扱う。文書集合 D は m 行 N 列の行列として表現できる。

文書検索で使う質問 $\boldsymbol{q}$ も、D の文書と同様のベクトルで表現する。

$$\boldsymbol{q} = (w_{1,q}, w_{2,q}, \ldots, w_{N,q})$$

2 つのベクトル $\boldsymbol{d_j}$ と $\boldsymbol{q}$ の類似度は、下記のベクトルの内積で定義する。

$$\text{類似度}(\boldsymbol{d_j}, \boldsymbol{q}) = \cos(\boldsymbol{d_j}, \boldsymbol{q}) = \frac{\sum_{i=1}^{N} w_{i,j} * w_{i,q}}{\sqrt{\sum_{i=1}^{N} w_{i,j}^2} * \sqrt{\sum_{i=1}^{N} w_{i,q}^2}}$$

最後に、検索結果を類似度の高い順に並び替えることで、質問 $\boldsymbol{q}$ に類似した文書 $\boldsymbol{d_j}$ のリストを作成する。

Sorensen-Dise 係数とその拡張モデル

これまでの研究で、コードの類似度に関し多くの提案がなされてきた。Baxter 氏らが開発した CloneDR [9] では、木構造で表現されたソースコード T1、T2 の間の類似度を以下の式で定義している。

類似度(T1, T2) = 2S / (2S + L + R)

S は T1 と T2 に共通するノードの数、L は T1 にだけ含まれるノードの数、R は T2 にだけ含まれるノードの数を示す。この類似度は、Sorensen-Dice 係数[22]を木構造のノードに適用したものと考えることができる。

Sorensen-Dice 係数は、2 個の集合 X_1 と X_2 の類似度を計算する指標であり、下記の式で定義されている。

$$\text{類似度}_{\text{Sorensen-Dice}}(X_1, X_2) = \frac{2|X_1 \cap X_2|}{2|X_1 \cap X_2| + |X_1 \cap \neg X_2| + |\neg X_1 \cap X_2|}$$

ここで $|X_1 \cap X_2|$ は、集合 X_1 と X_2 に共通する要素の数を示す。Sorensen-Dice 係数では、$|X_1 \cap X_2|$ に対して重み 2 を付けている。これは、要素が集合 X_1 と X_2 の両方に出現することを反映したものである。なお、$\neg X_1$、$\neg X_2$ は、それぞれ、X_1、X_2 の補集合を示す。

Sorensen-Dice 係数は、分類学で広く使われている Jaccard 係数と同類の指標である。ソフトウェア・クラスタリングでは、一般に、この 2 つの指標は他の指標よりも良い結果を導くことが知られている。ソフトウェア・クラスタリングでは、ある特徴を含むことで特徴の有無を判断することが多いためと考えられている。例えば、2 つのメソッドの類似度は、共通する変数やメソッドを参照していることで計算されるものであり、特定の変数やメソッドを使っていないことから計算されるものではない[22]。

本研究では、Sorensen-Dice 係数を類似度の定義の基盤とする。以下の式は、Sorensen-Dice 係数を n 個の集合に展開したものである。以降、この式を拡張 Sorensen-Dice 係数と呼ぶ。

$$\text{類似度}_{\text{Sorensen-Dice}}(X_1, X_2, \ldots, X_n) = \frac{n|X_1 \cap X_2 \cap \ldots \cap X_n|}{\sum_{r=0}^{n-1} |(n-r) SetComb(X_1 \cap X_2 \cap \ldots \cap X_n, r)|}$$

拡張 Sorensen-Dice 係数の分子は、集合 $X_1 \cap X_2 \cap \ldots \cap X_n$ に属する要素数に重み n を付けている。$|X_1 \cap X_2 \cap \ldots \cap X_n|$ は、$x_1 \in X_1, x_2 \in X_2, \ldots, x_n \in X_n$ である要素の組$<x_1, x_2, \ldots, x_n>$ の数を示す。

SetComb($X_1 \cap X_2 \cap \ldots \cap X_n$, r) は $\{X_1, X_2, \ldots, X_n\}$ から r 個を補集合に置き換えた積集合の組合わせを示す。拡張 Sorensen-Dice 係数の分母は、r = 0 から n−1 までの合計を計算している。この計算で SetComb($X_1 \cap X_2 \cap \ldots \cap X_n$, r) は、集合 $X_1, X_2, \ldots, X_n$ から、空集合を除く全ての組合せを生成する。重み n − r は、$X_1, X_2, \ldots, X_n$ で補集合に置き換えられていない集合の数を示す。

例えば、n = 3 の場合、集合は X_1、X_2、X_3 であり、拡張 Sorensen-Dice 係数の分子は $3|X_1 \cap X_2 \cap X_3|$ で計算される。また、拡張 Sorensen-Dice 係数の分母は、$3|X_1 \cap X_2 \cap X_3| + 2|X_1 \cap X_2 \cap \neg X_3| + 2|X_1 \cap \neg X_2 \cap X_3| + 2|\neg X_1 \cap X_2 \cap X_3| + |X_1 \cap \neg X_2 \cap \neg X_3| + |\neg X_1 \cap X_2 \cap \neg X_3| + |\neg X_1 \cap \neg X_2 \cap X_3|$ で計算される。

シーケンス構造派生検索モデル

ベクトル空間モデルおよび Sorensen-Dice 係数は、集合の類似度の指標であり、要素が出現する順番を反映していない。一方、ソースコードは本質的に一連の文字の列（シ

ーケンス）であり、文、ブロック、メソッド、クラスといった上位の概念で構造化される。このことを考慮すると、ソースコードを効率的に検索するためには、文、ブロックといった概念の列の概念を反映した類似度を使うことが必要である。

まず、シーケンスの概念の表記法を定義する。S_1 と S_2 を構造抽出ツールで抽出された文とする。$[S_1 \rightarrow S_2]$ で、S_1 に続いて S_2 が出現するシーケンスを表すものとする。一般に、任意の正数 n に対し、$[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n]$ で長さ n の文のシーケンスを表す。シーケンス構造派生検索モデル（以降、DSRM と略記する）の類似度は、形式的には、拡張 Sorensen-Dice 係数の ∩ 記号を→ 記号で置き換えることで定義される。

すなわち、任意の正数 m、n に対し、2 つの文のシーケンス $[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_m]$ と $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$ の類似度を、下記の式で定義する。

$$\text{DSRM 類似度}([S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_m], [T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n]) = \frac{n \mid [S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_m],\ [T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n] \mid}{\sum_{r=0}^{r=n-1} (n-r) \parallel [S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_m],\ SqcComb([T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n],\ r) \parallel}$$

ここで、一般性を失うことなく m≧n とすることができる。m＜n の場合は、$[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_m]$ と $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$ の位置を入れ替えるものとする。DSRM 類似度の分子、すなわち、$\mid [S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_m], [T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n] \mid$ は、j (0≦ j ≦m−n) に対し、$S_{j+1} = T_1, S_{j+2} = T_2, \ldots, S_{j+n} = T_n$ を満たす要素の数を示す。拡張 Sorensen-Dice 係数と同様に、長さ n のシーケンスに対し重み n を付けている。

DSRM 類似度の分母は、r = 0 から r = n−1 までのシーケンスの一致処理の繰り返しを示す。分母の "||" 記号は、シーケンスが一致する度に、$[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_m]$ の一致した要素を"n/a" 記号で書き換える。SqcComb ($[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$, r) 関数は、シーケンス $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$ から、r 個の要素に "¬" を付いたシーケンスの組合せを生成する。従って、重み n − r は、$[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$ で否定記号が付いていない要素の数を示す。

例えば、m = 5、n = 2 の場合、シーケンス $[A_1 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_2]$ と $[A_1 \rightarrow A_2]$ の DSRM 類似度は 0.5 になる。これは次の様に計算される。DSRM 類似度の分子は、$[A_1 \rightarrow A_2]$ が、$[A_1 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_2]$ に含まれるため、$2*\mid [A_1 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_2],\ [A_1 \rightarrow A_2] \mid = 2*1 = 2$ となる。

DSRM 類似度の分母は、まず r = 0 に対し、SqcComb($[A_1 \rightarrow A_2]$、 0) となり、シーケンス $[A_1 \rightarrow A_2]$を生成する。DSRM類似度の分子と同様に、$2*\parallel [A_1 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_2], [A_1 \rightarrow A_2] \parallel = 2*1 = 2$ となり、シーケンス$[A_1 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_2]$ は $[A_1 \rightarrow n/a \rightarrow n/a \rightarrow A_3 \rightarrow A_2]$ に書き換わる。

続いて、r = 1 に対し、SqcComb($[A_1 \rightarrow A_2]$, 1) は、 $[A_1 \rightarrow \neg A_2]$と $[\neg A_1 \rightarrow A_2]$ を生成する。$\parallel [A_1 \rightarrow n/a \rightarrow n/a \rightarrow A_3 \rightarrow A_2], [A_1 \rightarrow \neg A_2] \parallel$ の値は、A_1 が $[A_1 \rightarrow n/a \rightarrow n/a \rightarrow A_3 \rightarrow A_2]$に 1 個含まれるので 1 となり、シーケンス$[A_1 \rightarrow n/a \rightarrow n/a \rightarrow A_3 \rightarrow A_2]$ は $[n/a \rightarrow n/a \rightarrow n/a \rightarrow A_3 \rightarrow A_2]$ に書き換わる。最後に、$\parallel [n/a \rightarrow n/a \rightarrow n/a \rightarrow A_3 \rightarrow A_2], [\neg A_1 \rightarrow A_2] \parallel$ は、A_2 が $[n/a \rightarrow n/a \rightarrow n/a \rightarrow A_3 \rightarrow A_2]$に 1 個含まれるので 1 となり、シーケンス $[n/a \rightarrow n/a \rightarrow n/a \rightarrow A_3 \rightarrow A_2]$ は $[n/a \rightarrow n/a \rightarrow n/a \rightarrow A_3 \rightarrow n/a]$ に書き換わる。ここで、検索条件に含まれない A_3 は、DSRM 類似度に影響しない。

図 4 は、DSRM 類似度を計算するアルゴリズムの概略を示している。このアルゴリズムの先頭の関数 SimDSRM は、構造抽出ツールで取得したメソッド構造の集合 M と検索条件としてのシーケンス $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$ を入力とし、メソッド構造の集合に対する類似度の集合を戻り値として返す。図 4 の getMethodStructure(j) 関数は、j 番目（ j は任意の正数）のメソッド構造をシーケンスとして返す関数であり、メソッド構造の実装方式を抽象化している。

Count 関数は、DSRM 類似度の分子と分母を計算するためのシーケンスの一致数を計算するものである。Count 関数の引数は、メソッド構造 MS、検索条件としてのシーケンス TN、それに、否定記号で置き換えるシーケンスの要素数 R である。Count 関数の戻り値は、TN から R 個の要素を否定記号付に置き換えたシーケンスと一致する MS 内の要素の数である。

```
// "method_structure" のデータ型は "sequence" の集合.
// "sequence" は文の配列として表現されている.
// 入力: set_of_method_structure M;
// 入力: sequence [T1→T2→...→Tn];
// 出力: Sim[M.length]; 各メソッドに対するDSRM類似度
// --- SimDSRM 関数の定義
double[] SimDSRM( set_of_method_structure M, sequence [T1→T2→...→Tn] ){
 double Sim[M.length];    int Nume;    int Deno;
 for ( int j=0; j < M.length; j++ ) {
    Nume= Count( getMethodStructure(j), [ T1→T2→...→Tn], 0 );
    Deno= 0;
    for ( int r=1; r < [T1→T2→...→Tn].length; r++ ){
       Deno= Deno + Count( getMethodStructure(j), [ T1→T2→...→Tn], r );
    }
    if ( (Nume + Deno) == 0 ) { Sim[j]= -1; }
    else { Sim[j]= (double) Nume / (double)( Nume + Deno ); }
 }
 Return Sim;
}
// --- Count 関数の定義
int Count( method_structure MS, sequence TN, int R){
 statement[] S;        // Type of S is a "sequence" of statements
 statement[][] DS;     // Type of DS is a set of a "sequence" of statements
 statement[] SV;       // Type of SV is a "sequence" of statements
 int CT=0;
 // Generate derived sequence replacing R statements with negations
 DS= SqcComb( TN, R );
 for ( each S[] ∈ MS){
  for ( int j=1; j<=MS.length–TN.length; j++ ){
    for ( each SV[] ∈ DS ){
      for ( int k=1; k<= TN.length–R; k++ ){
        if ( S[j] = SV[k] for all k-th elements that are not negative )
              { CT= CT + (TN.length–R) };
      }
    }
  }
 }
Return CT;
}
```

図 4　DSRM 類似度を計算するアルゴリズムの概略

Count 関数の中で呼び出されている関数 SqcComb(TN、

R) は、DSRM 類似度の定義中の SqcComb 関数と同じ機能を提供するもので、検索条件であるシーケンス TN を構成する要素の内の R 個の要素を否定記号で置き換えたシーケンスの集合を返す関数である。

関数 SimDSRM は、メソッド構造の集合 M の各要素に対し、通常、0 以上 1 以下の値を返す。関数 SimDSRM が 1 になるのは、メソッド構造が検索条件としてのシーケンス $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$ だけを含み、シーケンス $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$ の部分シーケンスを含まない場合である。関数 SimDSRM が 0 になるのは、メソッド構造が検索条件としてのシーケンス $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$ を含まない場合である。なお、図 4 に示したアルゴリズムでは、DSRM 類似度の分母が 0 になる場合、戻り値 −1 を返す。

検索実験と評価

実験方法

Struts 2 を対象とした類似ソースコードの検索評価実験を行った。この実験は、(1)ソースコードに含まれる構造の探査、(2)探査された構造に関するソースコードの類似検索の順に実施した。類似検索は、ベクトル空間モデル、拡張 Sorensen-Dice 係数およびシーケンス構造派生検索モデル（DSRM）を用いて類似度の評価を行った。

構造の探査

ソースコードの類似検索を行う前に、対象とするソースコードに含まれる制御文とメソッド名の構造を探査する必要がある。本実験の目的から、先頭が特定の制御文である構造を探査することとした。この探査処理は、下記の式で表現でき、シーケンスに基づいた検索機能を流用することで実現できる。

$$\{ <[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n], |[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n]| > \mid \forall i \in [1..n] ([S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n] \in MS \wedge S1 = \text{"制御文の名称"}) \}$$

ここで MS は、Struts 2 を構成する 2,667 個のメソッドに対し、構造抽出ツールで抽出した制御文とメソッド名の構造の集合を示す。探査処理の結果、2,793 個のシーケンスが検出された。表 1 は、検出個数で上位 45 個のシーケンスを示しており、この 45 個のシーケンスに対してソースコードの類似検索を実施した。

ベクトル空間モデルによる検索結果

構造抽出ツールで抽出した制御文とメソッド名をベクトルの要素に割り付けることにより、メソッドをベクトルとして表現することができる。例えば、図 2 に示した *AbstractDirective* クラスの *putProperty()* メソッドは、(1, 1, 1, 1, 2, 1, 1) というベクトルで表現される。ここで、ベクトルの第 1 要素を *if* 文、第 2 要素を *else* 文、第 3 要素を *node.value*、第 4 要素を *Stirng.indexOF*、第 5 要素を *String.substring* などに対応させるものとする。このようにして、Struts 2.3.1.1 Core のソースコードは、1,420×2,667 の 2 次元行列として表現される。

表 1 検出数で上位 45 個のシーケンス

No	シーケンス	検出数
1	if{ → }	162
2	if{ → addParameter	102
3	if{ → addParameter → }	96
4	if{ → addParameter → } → if{	67
5	if{ → if{	59
6	if{ → } → }	47
7	if{ → Logger.debug	42
8	if{ → Logger.warn	41
9	if{ → Logger.warn → }	41
10	if{ → Logger.debug → }	36
11	if{ → Logger.warn → } → }	35
12	throw operationNotSupported → }	33
13	if{ → } → if{	32
14	else{ → }	27
15	if{ → try{	26
16	for{ → if{	23
17	if{ → } → else{	23
18	if{ → } → } → }	23
19	catch{ → Logger.error	20
20	else{ → } → }	20
21	catch{ → if{	19
22	if{ → for{	18
23	else_if{ → }	17
24	catch{ → Logger.error → }	17
25	if{ → } → if{ → }	15
26	if{ → StringBuilder.append	14
27	if{ → addParameter → } → }	14
28	if{ → String.substring	13
29	catch{ → if{ → Logger.warn	13
30	catch{ → if{ → Logger.warn → }	13
31	if{ → findValue	11
32	for{ → iterator.next	10
33	for{ → if{ → }	10
34	for{ → if{ → } → }	10
35	if{ → StringBuilder.append → }	9
36	else_if{ → addParameter → }	8
37	if{ → findValue → }	7
38	while{ → StringTokenizer.nextToken	6
39	if{ → addParameter → } → else{	6
40	while{ → Enumeration.nextElement	5
41	if{ → findString	4
42	for{ → List<String>.add	4
43	while{ → Iterator.next	4
44	if{ → findString → }	4
45	for{ → StringBuilder.append	3

String.substring メソッドは、特定の文字列をチェックした後に処理されることがあり、*if* 文の直後に比較的多く発生する。従って、ベクトル空間モデルによる検索実験では、*if* 文と *String.substring* メソッドが同数含まれることが妥当な検索条件である。*if* 文と *String.substring* メソッドが各 1 個含まれることを条件として検索した結果 387 個のメソッドが検索された。

表 2 は、ベクトル空間モデルの類似度の上位 29 個と 3 種類の検索モデルの類似度を示している。表 2 で、ベクトル空間モデルの類似度が最小なのは、9 行目の 0.190 であり、このメソッドは、他の 2 種類の検索モデルの類似度が 0.0 以上である（境界メソッド）。表 2 は、ベクトル空間モデルの類似度が境界メソッドの類似度（0.190）以上で

あるメソッドから構成されている。表 2 に示した 29 個のメソッドには、*if* 文に続いて *String.substring* メソッドが呼び出されていないものが含まれており、これらのメソッドは、表 2 の No 欄に網掛けを付けて表記している。

図 5 は、表 2 の No.12 に示した *ResourceUtil* クラスの *getResourceBase()* メソッドのソースコードである。このメソッドは、1 個の *if* 文と 1 個の *String.substring* メソッドを含んでおり、ベクトル空間モデルの類似度は 0.662 と、他のメソッドよりも類似度が高い。また、Sorensen-Dice 係数も 1.0 である。しかし、このメソッドは、*if* 文に続いて *String.substring* メソッドを呼び出していないので、シーケンス構造派生検索モデルの類似度は 0 である。文のシーケンスを考慮した場合、表 2 の No 欄に網掛けていないメソッドが、検索結果として相応しいメソッドである。

```
public static String getResourceBase(HttpServletRequest req) {
    String path = RequestUtils.getServletPath(req);
    if (path == null || "".equals(path)) {
      return "";
    }

    return path.substring(0, path.lastIndexOf('/'));
}
```

図 5 getResourceBase() メソッドのソースコード

拡張 Sorensen-Dice 係数による検索結果

検索条件として指定されたすべての要素が含まれていれば、拡張 Sorensen-Dice 係数の値は 0 以上になる。一方、ベクトル空間モデルでは、指定されたどれか一つの要素が含まれていれば、類似度が 0 以上になる。言い換えれば、拡張 Sorensen-Dice 係数では、検索条件としての要素の AND 条件を課しているのに対し、ベクトル空間モデルは要素の OR 条件を課している。従って、拡張 Sorensen-Dice 係数による検索結果は、ベクトル空間モデルの検索結果の部分集合である。このことは、表 2 の第 3、第 4 カラムに示した類似度で確認できる。例えば、No.19 に示した *StrutsConversionErrorInterceptor* クラスの *should AddError()* メソッドは、6 個の *if* 文を要素として含んでいるが、*String.substring* メソッドは含んでいない。このメソッドに対する拡張 Sorensen-Dice 係数は 0 であるが、ベクトル空間モデルの類似度は 0.307 である。

シーケンス構造派生検索モデルによる検索結果

検索条件として指定された要素のシーケンス [*if{*→*String.substring*] が含まれていれば、DSRM 類似度は 0 以上になる。この条件は、DSRM 類似度が、拡張 Sorensen-Dice 係数よりも厳しい条件を課していることを意味する。従って、DSRM 類似度による検索結果は、拡張 Sorensen-Dice 係数による検索結果の部分集合になる。このことは、表 2 の第 4、第 7 カラムに示した類似度で確認できる。例えば、図 5 に示した *ResourceUtil* クラスの *getResourceBase()* メソッド（表 2 の No.12 ）は、1 個の *if* 文と 1 個の *String.substring* メソッドを含んでいるので、拡張 Sorensen-Dice 係数は 1.0 である。一方、シーケンス [*if{*→*String.substring*] を含んでいないので、DSRM 類似度は 0 である。なお、図 3 に示

表 2　検索結果の上位 29 メソッド

No	メソッド名	ベクトル空間モデル	拡張Sorensen-Diceモデル			シーケンス構造派生検索モデル（DSRM）			ソース行数
		類似度	類似度	完全一致	部分一致	類似度	完全一致	部分一致	
1	MultiselectInterceptor::String intercept()	0.280	0.667	2	1	0.667	2	1	17
2	RegexPatternMatcher::RegexPatternMatcherExpression compilePattern()	0.204	0.667	2	1	0.667	2	1	23
3	AbstractDirective::void putProperty()	0.595	0.667	2	1	0.667	2	1	11
4	ServletUrlRenderer::String extractQueryString()	0.439	0.500	2	2	0.500	2	2	13
5	DefaultActionMapper::void handleSpecialParameters()	0.312	0.500	2	2	0.500	2	2	21
6	CheckboxInterceptor::String intercept()	0.329	0.400	2	3	0.400	2	3	24
7	RequestUtils::static String getServletPath()	0.673	0.667	4	2	0.333	2	4	19
8	Restful2ActionMapper::ActionMapping getMapping()	0.340	0.300	6	14	0.300	6	14	80
9	ServletUrlRenderer::void renderUrl()	0.190	0.286	2	5	0.286	2	5	48
10	ServletUrlRenderer::void renderFormUrl()	0.210	0.500	6	6	0.167	2	10	66
11	DefaultActionMapper..String getUriFromActionMapping()	0.358	0.286	4	10	0.143	2	12	43
12	ResourceUtil::static String getResourceBase()	0.662	1.000	2	0	0	0	2	7
13	TagUtils::static String buildNamespace()	0.300	0.667	2	1	0	0	3	16
14	PrepareOperations::String getUri()	0.474	0.667	2	1	0	0	3	12
15	DefaultActionMapper::String getUri()	0.435	0.667	2	1	0	0	3	13
16	RestfulActionMapper::ActionMapping getMapping()	0.304	0.333	2	4	0	0	6	36
17	Include::static String getContextRelativePath()	0.299	0.333	2	4	0	0	6	35
18	Component::String getComponentName()	0.347	0	0	1	0	0	1	6
19	StrutsConversionErrorInterceptor::boolean shouldAddError()	0.307	0	0	6	0	0	6	22
20	Component::String completeExpressionIfAltSyntax()	0.269	0	0	1	0	0	1	6
21	ApplicationMap::boolean equals()	0.269	0	0	1	0	0	1	7
22	RequestMap::boolean equals()	0.269	0	0	1	0	0	1	7
23	SessionMap<K;V>::boolean equals()	0.269	0	0	1	0	0	1	7
24	FileUploadInterceptor::boolean containsItem()	0.269	0	0	1	0	0	1	6
25	Counter::long getNext()	0.269	0	0	1	0	0	1	8
26	Counter::long getPrevious()	0.269	0	0	1	0	0	1	7
27	StrutsVelocityContext::boolean internalContainsKey()	0.211	0	0	6	0	0	6	24
28	SubsetIteratorTag::int doStartTag()	0.192	0	0	12	0	0	12	49
29	StrutsVelocityContext::Object internalGet()	0.190	0	0	6	0	0	6	23

した *putProperty()*メソッド（表 2 の No.3）の完全一致数が 2、部分一致数が 1 であり、DSRM 類似度は 0.667 となる。プログラムは、文のシーケンスで構成されている。DSRM 類似度は、文のシーケンスに基づいて検索するため、他の集合に基づく類似指標よりも高い選択率を有している。

検索結果の評価

表 1 に示した 45 個のシーケンスを検索条件とした実験を行った。ベクトル空間モデル、拡張 Sorensen-Dice 係数および DSRM 類似度によって検索されたメソッド数に基づいて検索結果を評価する。ベクトル空間モデルに対する DSRM 類似度の貢献度は、下記の式で計算する。

$$\frac{(\text{ベクトル空間モデルで検索されたメソッド数}) - (\text{DSRM で検索されたメソッド数})}{(\text{ベクトル空間モデルで検索されたメソッド数})}$$

表 2 に示した検索は表 1 の No.28 に該当するものであり、ベクトル空間モデルに対する DSRM 類似度の貢献度は、(29−11) / 29 = 62.1%である。

同様に、拡張 Sorensen-Dice 係数に対する DSRM 類似度の貢献度は、下記の式で計算する。

$$\frac{(\text{拡張 Sorensen-Dice で検索されたメソッド数}) - (\text{DSRM で検索されたメソッド数})}{(\text{拡張 Sorensen-Dice で検索されたメソッド数})}$$

表 1 の No.28 に該当する拡張 Sorensen-Dice 係数に対する DSRM 類似度の貢献度は、(17−11) / 17 = 35.3%である。

図 6 は、DSRM 類似度の貢献度をグラフで表現したものであり、横軸は表 1 の No カラムの番号に該当し、縦軸は貢献度を百分率（%）で示す。拡張 Sorensen-Dice 係数に対する DSRM 類似度の貢献度は 0% から 94.6%であり、ベクトル空間モデルに対する貢献度は 0% から 98.7%である。

DSRM 類似度の貢献度が 0%である表 1 の No.12 は、*throw operationNotSupported* という要素を含んでいる。この要素は、2,667 個のメソッドの内の 33 個のメソッドで各 1 回ずつ呼び出られている。このため、3 種類の検索方法で 33 個のメソッドを特定していることから、DSRM 類似度の貢献度は 0%となっている。検索条件 No.12 は、稀な検索条件と言える。

検索条件 No.32、No.38、No.40、No.43 は、*iterator.next*、*StringTokenizer.nextToken* といった稀にしか使われない要素から構成されている。これらの要素を含むメソッドでは、検索条件として与えられたシーケンスの形でしかこれらの要素が発生しなかったために、拡張 Sorensen-Dice 係数に対する DSRM 類似度の貢献度は 0%となった。これらの検索条件も稀な事例と言える。以上の 5 個の検索条件を除き、DSRM 類似度は、他の検索方法に比べて高い絞り込みを達成している。

先に述べたように、拡張 Sorensen-Dice 係数による検索結果は、ベクトル空間モデルの検索結果の部分集合である。しかし、上位の何件かを選択した場合は、この条件が満たされない場合がある。検索条件 No.4、No.33、No.34 では、ベクトル空間モデルの方が拡張 Sorensen-Dice 係数よりも高い絞込率を達成しているように見えるが、これは、No.4 の場合で上位 25 メソッド、No.33、No.34 で上位 45 メソッドを評価の対象としたために発生した事象である。

おわりに

ソースコードは文のシーケンスが本質的な意味を持つことから、一般の文書を対象としたベクトル空間モデルをはじめとする検索手法では検索精度が低いという問題点があった。本文では、ソースコードを構成する文のシーケンスに基づいたシーケンス構造派生検索モデル（DSRM）を提案し、Struts 2 Core を対象とした検索実験によって効

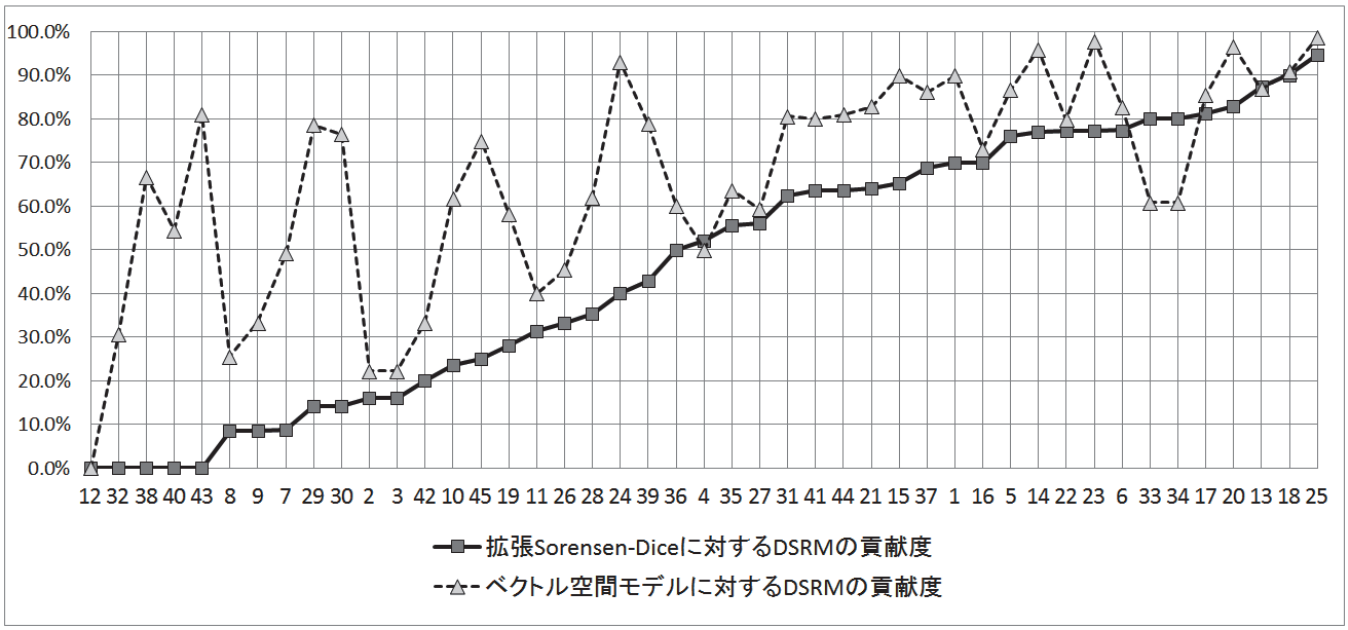

図 6　DSRM 類似度の貢献度

果を検証した。理論的には、DSRM による検索結果は、拡張 Sorensen-Dice 係数による検索結果の部分集合となる。また、拡張 Sorensen-Dice 係数の検索結果は、ベクトル空間モデルの検索結果の部分集合となる。Struts 2 Core を対象とした検索実験によって、特殊な事例を除けば、DSRM 類似度が他の 2 つの検索手法よりも高い絞込率を達成していることが確認できた。

今後は、メソッドの継承やオーバーローディングなどオブジェクト指向の機能を加味した検索機能の開発、開発環境と連動したユーザインターフェースの開発、さらに多くのオープンソースコードを使った実験を行い、実用性を検証する予定である。

謝辞

本文の初稿に対し，有益なご指摘をいただきました匿名の校閲者に感謝いたします．ご指摘いただいた事項は、本研究の的確な表現、および、本文の読み易さの向上に役立ちました。

参考文献

1) 情報処理推進機構（IPA)：ソフトウェア開発データ白書 2012-2013, 日経 BP 社 (2012).
2) Baker, B.S.: Parameterized Pattern Matching: Algorithms and Applications, Journal of computer and system sciences, 52, 1, pp.28-42(1996).
3) Mayrand, J., Leblanc, C., and Merlo, E.: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, In Proceedings of the 12th International Conference on Software Maintenance, pp.244-253 (1996).
4) Marcus, A. and Maletic, J.: Identification of High-level Concept Clones in Source Code, Proc. of the 16th International Conference on Automated Software Engineering, pp.107-114 (2001).
5) Higo,Y., Ueda, Y., Kusumoto, S., and Inoue, K.: Simultaneous Modification Support Based on Code Clone Analysis, Proc. of the 14th Asia-Pacific Software Engineering Conference, pp.262-269, (2007).
6) Kamiya,T., Kusumoto, S., and Inoue, K.: CCFinder: A Multi-linguistic Tokenbased Code Clone Detection System for Large Scale Source Code, IEEE Transactions on Software Engineering, Vol.28, No.7, pp.654-670 (2002).
7) イェユンウェン，中小路久美代： 文字と構造のサーチによるワンストップコード探索環境：CodeDepot, 情報処理学会デジタルプラクティス, Vol.2, No.2, pp.117-124 (2011).
8) Li, Z., Lu, S., Myagmar, S., and Zhou, Y. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, In IEEE Transactions on Software Engineering, Vol.32, 3, pp.176-192 (2006).
9) Baxter, I. D., Yahin, A., Moura, L. Sant'Anna, M., and Bier, L.: Clone detection using abstract syntax trees, Proc. of the 14th International Conference on Software Maintenance, pp.368-377 (1998).
10) Jiang, L., Misherghi, G., Su, Z., and Glondu, S.: DECKARD: Scalable and Accurate Tree-based Detection of Code Clones, In Proceedings of the 29th International Conference on Software Engineering, pp.96-105 (2007).
11) Komondoor, R. and Horwitz, S.: Using Slicing to Identify Duplication in Source Code, In Proceedings of the 8th International Symposium on Static Analysis, LNCS Vol.2126, pp.40-56 (2001).
12) Krinke, J.: Identifying Similar Code with Program Dependence Graphs, In Proceedings of the 8th Working Conference on Reverse Engineering, pp.301-309 (2001).
13) McCabe, T. J.: A complexty measure, IEEE Transactions on software engineering, Vol. SE-2, No.4, pp.308-320, 1976.
14) Chidamber, S. R. and Kemerer, C.F.: A Metrics Suite for Object Oriented Design, Software Engineering, IEEE Transactions on, Vol. 20, No. 6, pp.476-493(1994).
15) Mayrand, J., Leblanc, C., and Merlo, E.M.: Experiment on the Automatic Detection of Function Clones in a Software System using Metrics, the 12th International Conference on Software Maintenance (ICSM'96), pp.244-253(1996).
16) Patenaude, J-F, Merlo, E., Dagenais, M., and Lague, B.: Extending Software Quality Assessment Techniques to Java Systems, 7th International Workshop on Program Comprehension (IWPC'99), pp.49-56(1999).
17) Udagawa, Y.: An Approach to Retrieving Similar Source Codes by Control Structure and Method Identifiers, Lecture Notes in Computer ScienceVolume 7808, pp.252-259, (2013).
18) Gosling, J., Joy, B., Steele, G., and Bracha, G.: The Java Language Specification, Third Edition, ADDISON-WESLEY, (2005).
19) The Apache Software Foundation: About Apache Struts 2, http://struts.apache.org/release/2.3.x/.
20) Klaene, M.: Struts 1/Struts 2 Web アプリケーションフレームワークの比較, http://japan.internet.com/developer/20080304/26.
21) Salton, G. and Buckley, C.: Term-weighting approaches in automatic text retrieval, Information Processing and Management, Volume 24, Issue 5, pp.513-523 (1988).
22) Maqbool, O. and Babri, H.A.: Hierarchical Clustering for Software Architecture Recovery, IEEE Transactions on Software Engineering, Volume 33, Issue 11, pp.759-780 (2007) .