Vol.2013-IS-123 No.4
2013/3/15

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[†1]

ソースコードの確認作業を効率的に支援するために類似ソースコードの検索機能が必要とされている．本文では，制御文とメソッド名のシーケンスを検索条件とするソースコードの検索手法について論じている．本研究で開発した検索機能の特徴は，与えられた検索条件から有意な検索条件を派生させて検索を行うことである．この検索機能により，与えられた検索条件を完全に満たすソースコードだけでなく，部分的に満たすソースコードをも検索することができる．Struts 2 Core ソースコードを対象とした実験では，ベクトル空間モデルによる検索よりも，本研究で開発した検索機能が 16%から 94%の効率化を達成している結果を得た．

Applying Derived Structure Retrieval Model to Source Code Retrieval

Yoshihisa Udagawa[†1]

This paper presents an approach to improve source code retrieval using the structure of control statements. We develop a lexical parser that extracts structural information for each method including control statements and method identifiers. We present a source code retrieval model, named derived structure retrieval model, in which a retrieval conditions are defined as a sequence of control statements and/or method identifiers, and meaningful search conditions are derived from the given sequence. Experiments on the source code of Struts 2 Core show that the derived structure retrieval model outperforms the vector space retrieval model and it's variant by 16% to 94% in the number of retrieved methods.

1. はじめに

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ことから[1]，十分に実施されていない現状がある．開発フェーズでは，ソースコードの中身に精通した複数のプログラマによってソースコードが管理されるため，バグや設計変更に伴うソースコードの修正は通常，問題なく遂行される．一方，テストフェーズの後半になると，ソースコードの開発者とは異なる担当者が管理することになり，ソースコード修正コストを割高にする要因となっている．割高なコストを軽減する手段とし，統合ソフトウェア開発環境(IDE)でサポートされているソースコードの検索機能の活用が考えられるが，IDE におけるソースコードの検索機能は，単純な文字列による検索機能に留まっている[2][3][4].

これまでに，ソースコードの重複検出，流用検出およびソフトウェア保守を対象とした多数のソースコード検索手法が提案されている．これらは下記に大別される．

(1) テキスト比較

この方法は，ソースコードの対応する要素を文字列として比較するものである．Marcus 氏らは[13] 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．この方法は，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

ソースコードに含まれる識別名の違い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．この方法は，変数名の書き換えなどの影響を緩和したソースコードの比較が可能である[5] [11] [15].

(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)を使って比較を行う方法である[6][8]. 抽象構文木を使って類似コードの検出は，一般に計算コストが高い課題があり，抽象構文木の部分構造を特徴ベクトルで近似する研究も行われている[9].

(4) 学習モデルによる比較

バグレポートなどから修正すべきソースコードが抽出可能な場合，それらのソースコードの特徴をサポートベクターマシン(Support Vector Machines)などを使って学習させる．その学習結果に基づいて，ソースコードにバグが含まれる可能性を判定する．この手法は，修正すべきソースコード群が共通する特徴を有することを前提としている[10].

本文で述べる検索機能は，構造比較に基づく検索機能に関するものであり，"構造派生検索モデル"と命名されている．構造派生検索モデルの特徴は，制御文およびメソッド名から成るステートメントのシーケンスを検索条件とし，この検索条件から有意な検索条件を派生させ，与えられた

†1 東京工芸大学
Tokyo Polytechnic University

検索条件と派生させた検索条件のすべてに対して検索を実施することである．従って，検索結果は，指定された検索条件に合致する可能性のあるすべてのソースコードを含んでいる．“指定された検索条件に合致する可能性のあるすべてのソースコードを含む”という性質は，テスト段階に於いては発生したバグの横展開，ソフトウェアの受入れ検査時においてはコーディング規約に則した類似コードの網羅的な検索への応用に適した性質であると考えられる．

本文 2 章では，今回開発した検索機能の概要を述べる．3 章では，本研究で開発した構造抽出ツールの主な機能について述べる．4 章では，分析対象とした Java オープンソース Struts 2 と抽出した構造情報について述べる．5 章では，抽出した構造情報をベクトル空間モデルに適用して類似ソースコードを検索する手法と結果について述べる．6 章では，本研究で開発した構造派生検索モデルの原理と検索実験結果について述べる．7 章では，まとめと今後の研究方針について述べる．

2. 開発した検索機能の概要

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，処理速度を考慮し ANSI 規格準拠の C 言語で実装されている．

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．詳細は本文 5 章で述べる．

構造派生検索ツールは，制御文およびメソッド名を組合せたステートメントのシーケンス（並び）をキーにした検索を行う．指定されたキーに n 個の要素が含まれるとき，2n - 2 個の派生検索キーを生成し，可能性のあるすべてのステートメントのシーケンスに対して検索を実行することが特徴である．詳細は本文 6 章で述べる．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．

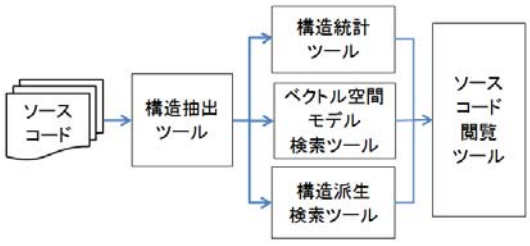

図 1　開発したツールの構成
Figure 1　Architecture of developed tools.

3. 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

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)

なお，クラス内で匿名クラス[7] 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)

(2) Java の制御文 [7]

下記の制御文をネスト構造とともに抽出する．

- if 文（else, else if 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
- try 文（catch，finally のバリエーションを含む）

(3) メソッドの中で呼び出しているメソッド名

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またはデータ型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ソースコードの違いを取り除いている．

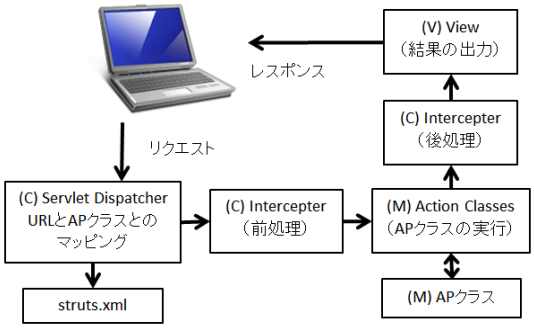

図 2　Struts 2 の概要
Figure 2　Overview of Struts 2.

4. Struts 2 ソースコードの構造抽出

4.1 Java フレームワーク Struts 2

Struts は 2000 年に開発者から Apache Software Foundation に寄贈され，以降，多くの業務アプリケーション開発の基盤として採用されてきた Java 向けの Web アプリケーションフレームワークである．Struts は Java ベースの Web アプリケーションを作成する上で人気の高いフレームワークであるが，システム開発者の要望に対応するために，Struts のアップデート版として Struts 2 を 2007 年にリリースした．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[12][16] ．

図 2 は，Struts 2 の動作概要と MVC モデルとの対応を図示している．コントローラ(C) は，Struts 2 の主要な機能であり，URL と AP クラスとのマッピング，HTTP リクエストなどの前処理，アプリケーションクラスの後処理，例外処理などを行う．モデル(M) は，アプリケーションで用意するクラスである．Struts 2 側でリクエストごとにオブジェクトの生成が行われるため，ロジックとデータが Action クラスにまとまったオブジェクト指向的な設計になっている．ビュー(V) は，アプリケーションクラスの実行結果を HTML ページに埋め込む処理を行う．デフォルトでは，HTML テンプレートとして JSP が利用されるが，FreeMarker，Velocity といったテンプレートも利用可能である．

4.2 Struts 2 からの構造抽出

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，前節で述べた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:	2,677 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 [14]:	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 3 は，Form クラスの evaluateClientSideJsEnablement 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．

```
Form::void evaluateClientSideJsEnablement(String, String, String)
#       7    1    22
{
    if{
    addParameter
    Configuration.getRuntimeConfiguration
    RuntimeConfiguration.getActionConfig
        if{
        ActionConfig.getInterceptors
            for{
                if{
                interceptorMapping.getInterceptor
                ValidationInterceptor.getExcludeMethodsSet
                ValidationInterceptor.getIncludeMethodsSet
                    if{
                    addParameter
                    }
                }
            }
        }
    }
}
```

図 3　構造の抽出結果の例

Figure 3　An example of extracted structure.

5.　ベクトル空間モデルによる類似検索

5.1 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．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 つのベクトルの類似度は，ベクトルの内積で定義する．$\boldsymbol{d_j}$ と $\boldsymbol{q}$ の類似度を下記の式で定義する．

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

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．

5.2 構造情報とベクトル

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 3 に示した Form::void evaluateClientSideJsEnablement メソッドは，

(4, 1, 2, 1, 1, 1, 1, 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，第 2 要素を for，第 3 要素を addParameter，第 4 要素を configuration.getRuntimeConfiguration などに対応させるものとする．

5.3 ベクトル空間モデルによる検索結果

AddParameter メソッドは，パラメータ名とその値を Struts 2 Core 内部で管理しているパラメータリストに追加する機能を提供しており，Struts 2 Core の多くのソースコードで利用されている．一般的な使い方としては，addParameter メソッドをパラメータに関する特定の条件を満たした直後に呼び出すため，if 文に続いて addParameter メソッドが呼び出されるシーケンスが散在する．今回の実験では，if 文と addParameter メソッドを含むことを検索条件として，ベクトル空間モデルによる類似検索を行った．

表 1 は，if 文と addParameter メソッドを同じ数だけ含むことを検索条件として，ベクトル空間モデルによって検索を行った結果の上位 27 個のメソッド名のリストである．28 位以下は，if 文のみの一致によって検索されたものであるため省略する．表 1 で網掛けを付けていないメソッドは，

表 1　ベクトル空間モデルで検索された上位 27 メソッド
Table 1　The top 27 retrieved methods by vector space model.

行番	メソッド名	ベクトル空間モデル	構造派生検索モデル			
		類似度	類似度	完全一致数	部分一致数	コード行数
1	Form::void evaluateExtraParams()	0.997	0.952	20	1	36
2	DoubleListUIBean::void evaluateExtraParams()	0.990	0.880	66	9	128
3	TextArea::void evaluateExtraParams()	0.985	1.000	8	0	15
4	Select::void evaluateExtraParams()	0.982	0.889	8	1	16
5	ComboBox::void evaluateExtraParams()	0.981	0.762	16	5	39
6	TextField::void evaluateExtraParams()	0.973	1.000	6	0	12
7	InputTransferSelect::void evaluateExtraParams()	0.967	0.737	14	5	50
8	UpDownSelect::void evaluateParams()	0.967	0.842	16	3	41
9	OptionTransferSelect::void evaluateExtraParams()	0.938	0.667	16	8	106
10	FormButton::void evaluateExtraParams()	0.913	0.200	2	8	31
11	ActionError::void evaluateExtraParams()	0.894	0.000	0	4	14
12	ActionMessage::void evaluateExtraParams()	0.894	0.000	0	4	14
13	ListUIBean::void evaluateExtraParams()	0.874	0.235	4	13	46
14	Checkbox::void evaluateExtraParams()	0.866	0.667	2	1	7
15	FieldError::void evaluateExtraParams()	0.866	0.667	2	1	6
16	Label::void evaluateExtraParams()	0.862	0.571	4	3	18
17	Form::void populateComponentHtmlId()	0.816	1.000	2	0	6
18	Password::void evaluateExtraParams()	0.816	1.000	2	0	6
19	Reset::void evaluateExtraParams()	0.816	1.000	2	0	5
20	Submit::void evaluateExtraParams()	0.816	1.000	2	0	5
21	StrutsVelocityContext::Object internalGet()	0.816	0.000	0	6	24
22	AdapterFactory::Node proxyNode()	0.816	0.000	0	5	13
23	StrutsVelocityContext::Object internalGet()	0.802	0.000	0	6	23
24	Form::void evaluateClientSideJsEnablement()	0.756	0.667	4	2	22
25	Token::void evaluateExtraParams()	0.707	0.000	0	6	21
26	File::void evaluateParams()	0.707	0.444	4	5	24
27	Anchor::void evaluateExtraParams()	0.707	1.000	4	0	14

表 2　Tf-idf ベクトル空間モデルでで検索された上位 25 メソッド
Table 2　The top 25 retrieved methods by vector space model using tf-idf.

行番	メソッド名	ベクトル空間モデル	構造派生検索モデル			
		類似度	類似度	完全一致数	部分一致数	コード行数
1	Form::void evaluateExtraParams()	0.994	0.952	20	1	36
2	DoubleListUIBean::void evaluateExtraParams()	0.987	0.880	66	9	128
3	ComboBox::void evaluateExtraParams()	0.977	0.762	16	5	39
4	Select::void evaluateExtraParams()	0.975	0.889	8	1	16
5	TextArea::void evaluateExtraParams()	0.967	1.000	8	0	15
6	TextField::void evaluateExtraParams()	0.943	1.000	6	0	12
7	Checkbox::void evaluateExtraParams()	0.932	0.667	2	1	7
8	FieldError::void setFieldName()	0.928	0.000	0	1	3
9	InputTransferSelect::void evaluateExtraParams()	0.913	0.737	14	5	50
10	UpDownSelect::void evaluateParams()	0.907	0.842	16	3	41
11	FieldError::void evaluateExtraParams()	0.858	0.667	2	1	6
12	ActionError::void evaluateExtraParams()	0.846	0.000	0	4	14
13	ActionMessage::void evaluateExtraParams()	0.846	0.000	0	4	14
14	OptionTransferSelect::void evaluateExtraParams()	0.838	0.667	16	8	106
15	ListUIBean::void evaluateExtraParams()	0.811	0.235	4	13	46
16	Label::void evaluateExtraParams()	0.747	0.571	4	3	18
17	FormButton::void evaluateExtraParams()	0.742	0.200	2	8	31
18	Password::void evaluateExtraParams()	0.686	1.000	2	0	6
19	Reset::void evaluateExtraParams()	0.686	1.000	2	0	5
20	Submit::void evaluateExtraParams()	0.686	1.000	2	0	5
21	Token::void evaluateExtraParams()	0.679	0.000	0	6	21
22	Form::void populateComponentHtmlId()	0.510	1.000	2	0	6
23	Anchor::void evaluateExtraParams()	0.509	1.000	4	0	14
24	File::void evaluateParams()	0.507	0.444	4	5	24
25	Form::void evaluateClientSideJsEnablement()	0.472	0.667	4	2	22

if 文あるいは addParameter メソッドを要素として含むが，if 文に続いて addParameter が存在するというシーケンスを含まないメソッドを示している．言い換えると，構造派生検索モデルで完全一致数が 0 であるメソッド名である．

表 2 は、tf-idf 法によって重み付けを行ったベクトル空間モデルによって検索を行った結果の上位 25 個のメソッド名のリストである．26 位以下は，if 文のみの一致によって検索されたものであるため省略する．

表 1 と同様、表 2 で網掛けを付けていないメソッドは，if 文あるいは addParameter メソッドを要素として含むが，if 文に続いて addParameter が存在するというシーケンスを含まないメソッドを示している．

詳細は 6 章で述べるが、多くの場合で、tf-idf 法を用いた重み付けを行った方が、tf-idf 法を用いないベクトル空間モデルよりも良い結果になる。しかしその貢献度の増加は、多くの場合で 10%程度であり、tf-idf 法による本質的な改善は認められない．

6. 構造派生検索モデルによる類似検索

6.1 構造派生検索モデル

ベクトル空間モデルでは，検索キーとなる単語の種類と個数を手掛かりに, 類似する文書ベクトルを検索していた．言い換えると，ベクトル空間モデルは，検索キーとなる単語の種類の集合，あるいは，“組合せ”に対する検索を実施している．これに対し，プログラムはステートメントのシーケンス（並び）が本質的な役割を果たす．

構造派生検索モデルは，ステートメントのシーケンスを検索条件としている．まず，条件として与えられたシーケンスで検索する．続いて，そのシーケンスの任意の要素をワイルドカードで置き替えたシーケンスを派生させ，派生させたすべてのシーケンスで検索を行うものである．

S_1，S_2 をプログラムのステートメントとしたとき，S_1 に続いて S_2 が出現するステートメントのシーケンスを $[S_1 \rightarrow S_2]$ と表記する．一般に，n を任意の正の整数とし，S_i $(1 \leqq i \leqq n)$ をステートメントとするとき, n 個のステートメントのシーケンスを $[S_1 \rightarrow S_2 \rightarrow ... \rightarrow S_n]$ と表記する．

シーケンス$[S_1 \rightarrow S_2 \rightarrow ... \rightarrow S_n]$ に類似するプログラムを網羅的に検索する方法として，本研究では下記の方法を採用した．

(1) シーケンス$[S_1 \rightarrow S_2 \rightarrow ... \rightarrow S_n]$ に一致するものを検索する．

(2) n≧2 の場合, シーケンス$[S_1 \rightarrow S_2 \rightarrow ... \rightarrow S_n]$ から 1 個..., n-1 個の要素を取り除いたシーケンスに一致するものを

検索する．

図 4 は，シーケンスを構成する要素が 3 個の場合の，要素の包含関係に基づくシーケンスの構造を示している．シーケンス [*→ *→ *] は，任意のシーケンスと一致するので，検索条件としては機能しない．一般に，n 個の要素から成るシーケンスからは，2^n - 2 個の有効な検索条件が派生し，検索条件として与えられたシーケンスと合わせて 2^n - 1 個の有効な検索条件を得ることができる．構造派生検索モデルは，この 2^n - 1 個のシーケンスに対して検索を行うため，シーケンスを構成する要素の 1 つでも一致するものがあれば，検索結果に反映される．

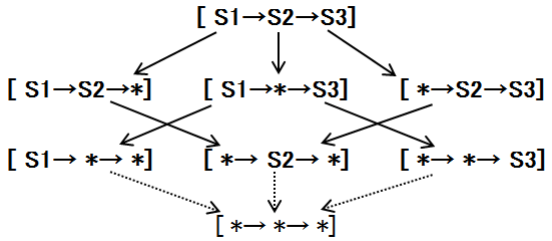

図 4　3 個の要素からなるシーケンスと派生シーケンス

Figure 4　A three-element sequence and derived sequence.

図 5 は，シーケンス[S_1→S_2→...→S_n]を検索キーとした構造派生検索による類似度計算アルゴリズムである．ここで，m をメソッドの構造情報とし，r を 1 以上 n 以下の値を取る変数 (1≦r≦n) とする．Count(m, [S_1→S_2→...→S_n], r) を，シーケンス[S_1→S_2→...→S_n] から r 個の要素をワイルドカード “*” で置き換えたシーケンスに一致する部分構造情報の数を返す関数とする．また，getMethodStructure(j)を検索対象とするメソッドの構造情報リストの j 番目のメソッドの構造情報を取り出す関数とする．

この定義によれば，シーケンス[S_1→S_2→...→S_n] だけから構成されるソースコードは類似度が 1.0 になる．また，シーケンス[S_1→S_2→...→S_n] を含まない場合は，類似度が 0.0 になる．

```
Input:   set_of_structure M;
Input:   sequence [S1→S2→...→Sn];
Output: Sim[M.length];
  //   Declare array Sim[].
double Sim[M.length];
  //   Compute Similarity for each method in M.
int Nume;   int Deno;
for ( int j=0; j < M.length; j++ ) {
  Nume= Count(getMethodStructure(j),[S1→S2→...→Sn],0);
  Deno= 0;
  for ( int r=1; r < [S1→S2→...→Sn].length; r++ ){
    Deno= Deno
          + Count(getMethodStructure(j),[S1→S2→...→Sn],r);
  }
  Sim[j]= (double) Nume / (double)( Nume + Deno );
}
```

図 5　シーケンスを使った検索の類似度計算法

Figure 5　Algorithm to compute the similarity of methods for a sequence [S_1→S_2→...→S_n]

6.2 構造派生検索モデルによる検索結果

5.3 節のベクトル空間モデルによる検索に対応する実験として，if 文の直後に addParameter メソッドが呼び出されることを検索条件とした検索を行った．検索であるシーケンス[if→addParameter] を少なくとも 1 個含むメソッドは，表 1 で網掛けを付けた 21 個のメソッドである．表 1 で網掛

表 3 検索モデルと貢献度

Table 3　Degree of improvement for the models.

行番	検索パターン	メソッド数 (VSM)	メソッド数 (Tf-idf VSM)	メソッド数 (DSR)	貢献度 (対VSM)	貢献度 (対Tf-idf VSM)
1	if{ → addParameter	27	25	21	22.2%	16.0%
2	if{ → Logger.debug	72	61	31	56.9%	49.2%
3	if{ → Logger.warn	45	32	13	71.1%	59.4%
4	while{ → StringTokenizer.nextToken	16	8	6	62.5%	25.0%
5	while{ → Enumeration.nextElement	13	11	5	61.5%	54.5%
6	while{ → Iterator.next	28	21	4	85.7%	81.0%
7	for{ → iterator.next	21	13	9	57.1%	30.8%
8	for{ → List<String>.add	5	6	4	20.0%	33.3%
9	for{ → StringBuilder.append	10	12	3	70.0%	75.0%
10	catch{ → Logger.error	41	43	18	56.1%	58.1%
11	if{→ addParameter → }	28	26	21	25.0%	19.2%
12	if{→ Logger.warn → }	49	21	3	93.9%	85.7%
13	if{→ Logger.debug → }	98	38	26	73.5%	31.6%
14	if{→ } → }	154	136	9	94.2%	93.4%
15	if{→ } → if{	206	175	22	89.3%	87.4%
16	if{→ } → else{	192	151	22	88.5%	85.4%
17	for{→ if{→ }	30	23	9	70.0%	60.9%
18	if{→ addParameter → } → else{	14	19	4	71.4%	78.9%
19	if{→ Logger.warn → } → }	58	40	24	58.6%	40.0%
20	if{→ } → } → }	247	197	18	92.7%	90.9%
21	for{→ if{→ } → }	13	23	9	30.8%	60.9%

けを付けていない 6 個のメソッドは，if 文あるいは addParameter メソッドを含むが，シーケンス[if→addParameter] を含んでいない．ベクトル空間モデルでは，if 文あるいは addParameter メソッドの出現個数を基準にして類似度を評価するために，検索されたメソッドの類似度は 0.707～0.894 という比較的高い値になっている．一方，ソースコードを検索する立場からは，シーケンス[if→addParameter] を含むことを期待することから，構造派生検索モデルはベクトル空間モデルによるものよりも，効率的に条件を満たすメソッド名を検索している．効率化の貢献度は，ベクトル空間モデルが 27 個に対し，構造派生検索モデルは 21 個であることから，(27-21) / 27= 22.2% となる．

表 3 は，ベクトル空間モデル，および，tf-idf 重み付けを行ったベクトル空間モデルに対する構造派生検索モデルの貢献度を 21 個の検索パターンについてまとめたものである．検索パターンによってばらつきがあるが，ベクトル空間モデルに対しては 20%から 94%, if-idf 重み付けを行ったベクトル空間モデルに対しては 16%から 93%の効率化を達成している．

6.3 処理時間の計測結果

表 4 は，ベクトル空間モデル (VSM)，tf-idf 重み付けを行ったベクトル空間モデル (Tf-idf VSM) および構造派生検索モデル (DSR) の処理時間を示している．実験環境は下記の通りである．

CPU: Inter Core i3 540 3.07GHz
メモリ: 4.00GB
OS: Windows 7 64Bit
表計算ソフト: Office 2010 Excel

Struts 2.3.1.1 Core ソースコードから抽出した制御文とメソッド名の重複を除いた数は 1,420 個である．メソッドが 2,677 個あるため，VSM および tf-idf VSM による検索のために 1,420×2,677 のマトリックスを作成し，Excel シート 1 枚に格納した．また，DSR で使用するメソッドごとの構造情報も 1 枚の Excel シートに格納した．主要なデータはセルから取得する構造であり，この点では，DSR，VSM および tf-idf VSM とも差異がない．

表 4 に示したように, VSM での検索時間は平均で 36.202 秒, tf-idf VSM で 92.428 秒である．Tf-idf VSM による検索では，制御文とメソッド名 1,420 個の tf-idf を検索の都度計算している. この処理では 18,441,740 回のセルへのアクセスが発生し，約 56 秒を要している．VSM と tf-idf VSM では約 56 秒の差異があるため，類似度を計算する処理としては，VSM と tf-idf VSM でほぼ同じ処理時間を要していることが分かる．

長さ 2 のシーケンスを検索キーとした実験では，DSR の性能は，平均で 0.512 秒である．長さ 2 のシーケンスでは，検索条件として与えられたシーケンスと 2 個の派生シーケンスに対する合計 3 回の検索を行うため，1 回の平均の処理時間は 0.171 秒である．長さ 3 のシーケンスを検索キーとした実験では合計 7 回の検索を行い，平均で 1.312 秒を要した．1 回の検索で要した処理時間は 0.187 秒である．長さ 4 のシーケンスを

表 4 検索時間の測定結果（単位はミリ秒）
Table 4 Results of performance measurement (Time in milliseconds)

行番	検索パターン	VSM	Tf-idf VSM	DSR
1	if{ → addParameter	35553	91308	546
2	if{ → Logger.debug	36988	92087	499
3	if{ → Logger.warn	36005	92025	500
4	while{ → StringTokenizer.nextToken	36145	92758	515
5	while{ → Enumeration.nextElement	36161	94115	515
6	while{ → Iterator.next	35943	93226	500
7	for{ → iterator.next	36083	91978	515
8	for{ → List<String>.add	36551	92290	515
9	for{ → StringBuilder.append	35880	92118	514
10	catch{ → Logger.error	36482	91760	499
11	if{ → addParameter → }	36645	92836	1124
12	if{ → Logger.warn → }	37222	91619	1292
13	if{ → Logger.debug → }	36551	91635	1123
14	if{ → } → }	36474	93024	1388
15	if{ → } → if{	36364	92306	1388
16	if{ → } → else{	36582	92945	1404
17	for{ → if{ → }	36286	93055	1466
18	if{ → addParameter → } → else{	35662	92571	2824
19	if{ → Logger.warn → } → }	36489	92321	2839
20	if{ → } → } → }	35974	91682	2908
21	for{ → if{ → } → }	34211	93335	3034

検索キーとした実験では，合計 15 回の検索を行い，平均で 2.901 秒を要した．1 回の検索で要した処理時間は 0.193 秒である．

検索処理に関わるオーバーヘッドがあるため，シーケンスが長くなるほど 1 回の検索に要する時間が 3%から 8%程度増加していることが判明した．

DSR と VSM による検索を比較すると，DSR が 1 桁程度高速である．VSM で使用したマトリックスの要素値の大部分は 0 であることを考慮すると，実装方法の工夫により VSM の高速化が期待できるが，本研究の主旨から外れるので，以上の比較に留める．

7. まとめと今後の研究方針

類似ソースコードの検索機能は，プログラミングの効率化だけでなく高品質なソフトウェアを開発・管理するために必要となる機能である．一方，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．本研究で開発した構造派生検索モデルは，プログラムのステートメントのシーケンスを検索キーとし，そのシーケンスから有意な検索シーケンスを派生させることにより，検索キーとして与えられたシーケンスを部分的に含むメソッドを網羅的に検索する特徴がある．Struts 2 Core を対象とした検索実験では，構造派生検索モデルはベクトル空間モデルと比較して 16%から 94%の検索効率の向上を達成していることが確認できた．Visual Basic による実装では，構造派生検索モデルはベクトル空間モデルと比べて十分な処理性能を有することが確認できた．今後は，構造派生検索機能を充実させ，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，構造派生検索機能の特質を明らかにする予定である．

参考文献

1) 情報処理推進機構（IPA）：ソフトウェア開発白書 2012,日経 BP 社,2012.
2) Eclipse: Featured Eclipse Project, http://www.eclipse.org/, 2012.
3) エンバカデロ: C++Builder, http://www.embarcadero.com/jp/, 2012.
4) マイクロソフト: Visual Studio, http://www.microsoft.com/ja-jp/dev/, 2012.
5) B.S. Baker: Parameterized pattern matching: algorithms and applications. Journal Computer System Science vol.52, no.1, pp.28-42, February 1996.
6) I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier: Clone detection using abstract syntax trees. Proc. of the 14th International Conference on Software Maintenance, pp.368-377, November 1998.
7) J. Gosling, B. Joy, Guy Steele, G. Bracha: The Java Language Specification, Third Edition, ADDISON-WESLEY, May 2005.
8) S. Horwitz, T.W. Reps, and D. Binkley: Interprocedural slicing using dependence graphs, ACM Trans. on Programming Lang. and Sys. vol.12, no.1, pp.1-34, January 1990.
9) L. Jiang, G. Misherghi, Z. Su, and S. Glondu: DECKARD: Scalable and accurate tree-based detection of code clones, Proc. of the 29th international conference on Software Engineering, pp.96-105, May 2007.
10) K. Jindarak, and U. Sammapun: Fault Prediction Using the Similarity Weights Method, Proc. of International Conference on Modeling, Simulation and Control, pp.173-178, October 2011.
11) T. Kamiya, S. Kusumoto, and K. Inoue: 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, July 2002.
12) M. Klaene: 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.html, March 2008.
13) A. Marcus, and J. Maletic: Identification of high-level concept clones in source code. Proc. of the 16th international Conference on Automated Software Engineering, pp.107-114, November 2001.
14) T.J. McCabe: A complexity measure, IEEE Transactions on software engineering, vol.2, no.4, pp.308-320, December 1976.
15) E. McCreight: A space-economical suffix tree construction algorithm. Journal of the ACM vol.23, no.2, pp.262-272, April 1976.
16) Rose India Technologies: Struts Tutorials - Jakarta Struts Tutorial, http://www.roseindia.net/struts/struts2/index.shtml, 2008.