Retrieving Similar Source Codes by Control Structure Metrics

Yoshihisa Udagawa[†1]

In this paper, we present an approach to improve source code retrieval using the structure of control statements. We develop a lexical parser and extract structural information, which is then converted into a document vector used for information retrieval. We show that the number of control statements largely depends on cyclomatic complexity. Next we employ a difference measurement, which is the Euclidean distance between two vectors, to improve the vector space model used for retrieving source codes. Finally, we conduct two types of experiments using the open source Struts 2 Core. In the first experiment, we use the try-catch and synchronized statements as keys, and examine the quality of the code retrieved with respect to exceptions and thread control. In the second experiment, we retrieve code on the basis of similarity and difference measurements. In both experiments, several sets of source codes that are presumably maintained in a consistent manner are retrieved.

1. Introduction

Numerous open source programs are available [1][13][14] for the development of Web applications for industrial use and for educational purposes in advanced programming courses. However, many valuable programming techniques available in open-source programs remain unexploited. The aim of our work is to search for excellent source codes that have a given control structure. Specifically, we develop sophisticated techniques to retrieve similar source codes using the structural information of control structures, including conditional, iteration, and exception handling statements.

Various techniques have been proposed to collect similar source codes, especially in the field of software clone detection. These techniques can be classified into four categories:

(A) Text-based comparison

This approach compares source codes in the same partition. Marcus et al. [8] compare pieces of text identifiers using a latent semantic indexing technique developed for information retrieval. The key idea of this approach is to identify source-code fragments using similar names or identifiers.

(B) Token comparison

In this approach, before comparison, tokens of identifiers (data type names, variable names, etc.) are replaced by special tokens, and then similar subsequences of tokens are identified [6]. Because the encoding of tokens abstracts from their concrete values, code fragments that are different only in parameter naming can be detected. McCreight [11] and Baker [2] show that a suffix tree of tokens can be built in linear time and space with respect to the input length. This tree results in fair performance when comparing large-scale source codes.

(C) Metrics comparison

This approach characterizes code fragments using different metrics, and compares these metric vectors instead of directly comparing the code [9]. To detect similar codes, the Euclidean distance for these metric vectors is used. In addition, metrics comparison techniques are proposed for detecting duplicated Web pages [7].

(D) Structure-based comparison

This approach applies pattern matching and complex algorithms on abstract syntax trees or dependency graphs. Baxter et al. propose a method using abstract syntax trees for detecting exact and near-miss program source fragments [3]. Horwitz et al. propose a method that generates a slice of an entire program in a system dependence graph [4]. However, the processing of structure-based comparison is computationally more expensive. Thus these techniques do not scale to large code bases. Jiang et al. developed an algorithm that characterizes a sub-tree using a vector, whose elements represent the number of occurrences of a specific tree pattern in the sub-tree. Specifically, they propose an algorithm that characterizes sub-trees using numerical vectors, and clusters these vectors based on their Euclidean distances [5].

Our approach is a combination of the structure-based comparison and metrics comparison. First, we developed a lexical parser and extracted structures of source codes for control statements, such as if-else, for and try-catch. Then, we inputted the extracted structural information to the vector space model and computed a similarity measure, which was used to find similar methods in Java. Next, we applied our retrieval methods to the source codes of Struts 2 Core. Struts 2 Core was selected because it is widely used to develop Web applications for both industrial and educational use, and its size is appropriate for this case study.

The rest of this paper is organized as follows. In Section 2, we present an overview of our approach. Specifically, we describe the system we developed, the structure metrics used, and the statistical results of the structure metrics obtained from the Struts 2 Core source codes. In Section 3, we discuss how source code can be retrieved using a specific control structure. In Section 4, we discuss a similarity-retrieval approach based on a vector space model that uses the structure metrics. Section 5 concludes the paper.

2. Overview

2.1 Complexity metrics and control structure

To characterize the different facets of software complexity, several metrics can be used, such as file level metrics, object-oriented metrics, and complexity metrics for program

†1 Tokyo Polytechnic University

modules. Cyclomatic complexity [10] is defined on the basis of graph-theoretic properties, i.e. , "Edges - Nodes + Connected Components," and is widely used to estimate the difficulty associated with testing or planning a testing strategy.
Cyclomatic complexity is approximately equal to the number of control statements or decision points (if-then-else, for loop, while loop, etc.) contained in a program. This metric does not consider the function of the control statements. Thus, when retrieving source codes with the same control structure, using this approximation metric is considered an oversimplification.

2.2 Tools Developed

Figure 1 illustrates a high-level architecture of the tools we developed. The structure extraction tool is implemented in C-language and is used to extract control structures of Java programs placed in a given directory. The structure extraction tool extracts code structures from every method of a class in Java. Then, these extracted structures are inputted to the statistic tool, structure analysis tool, and retrieval tool, which are written in VB. Finally, the outputs of these modules are fed into the source code viewer. In our current implementation, the tools written in VB and the source code viewer are manually connected.

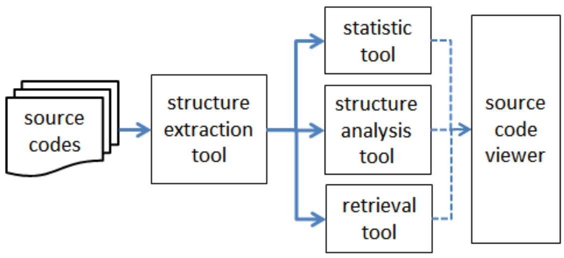

Figure 1. High-level architecture of tools developed

2.3 Struts 2 Core and its file metrics

In general, a framework automates common tasks, and thereby providing a user platform that simplifies web development. The Struts 2 Core framework implements the model-view-controller (MVC) design pattern. Table 1 summarizes the package structure of Struts 2 Core.

In the MVC design pattern, the controller receives inputs and then maps user requests to appropriate actions. In Struts 2 Core, the classes in the dispatcher package perform the tasks of the controller. The model in MVC is responsible for maintaining the data of the application or business logic. It also validates data entered by the user. The maintained data is returned to the controller. The action component class in the components package mainly implements the model in MVC. When the controller triggers the view in MVC, it presents the data in a particular format. In Struts 2 Core the view is mostly implemented by the classes in the freemarker, jsp, and velocity packages.

We can estimate the volume of the source codes using file metrics. Table 2 summarizes Typical file metrics for important packages. Struts 2 Core consists of 46,100 lines in source code. As for the number of lines, Struts 2 Core is a middle scale application in industry. The number of Java files differs from the number of declared classes because some java files include definitions of inner classes and anonymous classes.

Table 1. Package structure of Struts 2 Core

Directory Path	No. Files
struts2	5
struts2/components	57
struts2/components/template	8
struts2/config	10
struts2/dispatcher	20
struts2/dispatcher/mapper	8
struts2/dispatcher/multipart	3
struts2/dispatcher/ng/filter	4
struts2/dispatcher/ng/listener	2
struts2/dispatcher/ng/servlet	2
struts2/dispatcher/ng	5
struts2/impl	3
struts2/interceptor	27
struts2/interceptor/debugging	3
struts2/interceptor/validation	2
struts2/servlet/interceptor	1
struts2/util	29
struts2/views	3
struts2/views/annotations	3
struts2/views/freemarker	5
struts2/views/freemarker/tags	45
struts2/views/jsp	20
struts2/views/jsp/ui	34
struts2/views/jsp/iterator	5
struts2/views/util	3
struts2/views/velocity	3
struts2/views/velocity/components	39
struts2/views/xslt	19
Total	368

```
PrefixBasedActionMapper::getMapping
{
    for
        if
            if
            }
            if
                if
                    if
                        for
                            if
                            }
                            else if
                            }
                            else if
                            }
                            else
                            }
                        }
                    }
                }
            }
            else if
            }
        }
    }
    if
    }
}
```

Figure 2. Control structure of the maximum nesting level 7

2.4 Structural metrics of Struts 2 Core

We used the structure extraction tool and extracted approximately 12,700 lines of code of control structures. Table 3 summarizes the statistics of the extracted control structures. The statistics indicates that the top six extracted statements are if, else, try, catch, for, and else-if statements. Note that only two do-while statements are used in the Struts 2 Core source codes.
Table 4 lists the top six methods in terms of cyclomatic complexity. Cyclomatic complexity is approximately proportional to the number of control statements. From Table 4 we see that the if statements are the main contributors of cyclomatic complexity. In software engineering, it is recommended to maintain cyclomatic complexity under 10. Thus, it is suggested that complex methods are recommended to be separated into two or more methods. The simplification of complex methods is beyond the scope of this study and thus not addressed here.

A maximum nesting level of 7, with cyclomatic complexity 13, is recorded in the getMapping method in the PrefixBasedActionMapper.java file in the org.apache.struts2.dispatcher.mapper directory. Figure 2 illustrates the extracted control structure of the getMapping method.

3. Code Retrieval Using a Specific Control Structure

3.1 Try-catch-finally statement

For developers and students, the fastest way to learn how to accomplish a programming task is to look at an example of a similar implementation. During maintenance tasks, engineers spend the majority of their time identifying code statements related to a bug, and finding similar codes that may cause the same bug. Code retrieval methods allow engineers to explore source codes in a quicker and deeper manner.

Table 2. Typical file metrics for important packages

	<top>	components	config	dispacher	impl	intercepter	util	views	Total
CountJavaIFile	5	65	10	44	3	33	29	179	368
CountDeclClass	5	76	13	57	3	38	36	186	414
CountDeclFunction	19	771	64	372	6	156	188	1,101	2,677
CountLine	620	12,213	1,500	8,097	210	4,904	3,162	15,394	46,100
CountLineBlank	120	1,726	179	974	31	522	515	2,581	6,648
CountLineCode	178	5,948	636	3,802	105	1,946	1,554	7,374	21,543
CountLineComment	323	4,546	693	3,328	74	2,440	1,095	5,455	17,954
CountStmtDecl	120	2,207	247	1,434	39	775	586	3,385	8,793
CountStmtExe	27	1,930	221	1,363	27	717	570	2,212	7,067
RatioCommentToCode	1.81	0.76	1.09	0.88	0.70	1.25	0.70	0.74	0.83
RatioBlankToCode	0.67	0.29	0.28	0.26	0.30	0.27	0.33	0.35	0.31

Table 3. Statistics of the extracted control structures

Number of statements	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…	34	35	36	…	50	51	Net
synchronized	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0		0	0	8
try	80	14	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0		0	0	114
catch	61	10	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0		0	0	100
final	26	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0		0	0	32
if	200	74	42	24	11	17	6	3	4	2	4	1	2	1	3	0	2	0	…	0	1	0	…	0	1	1110
else	94	20	8	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0		0	0	166
else if	18	8	8	1	2	2	1	0	0	0	0	0	0	0	0	0	0	0		0	0	0		0	0	91
for	72	8	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0		0	0	98
while	29	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0		0	0	31
do-while	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0		0	0	2
																									Total	1752

Table 4. The top six methods complexity

No	Cyclomatic Complexity	No. of Control Statements	No. of if	Package_name.Class_name.Method_name
1	55	57	51	org.apache.struts2.components.UIBean.evaluateParams
2	41	43	35	org.apache.struts2.components.DoubleListUIBean.evaluateExtraParams
3	25	26	17	org.apache.struts2.dispatcher.mapper.Restful2ActionMapper.getMapping
4	22	26	17	org.apache.struts2.views.velocity.VelocityManager.loadConfiguration
5	18	21	15	org.apache.struts2.views.util.DefaultUrlHelper.buildUrl
6	18	20	15	org.apache.struts2.views.jsp.iterator.SubsetIteratorTag.doStartTag

Table 5. Methods containing one try statement and three or four catch statements

No	Package Name	File / Class Name	Method Name	Exception Handling
1	org.apache.struts2.dispatcher.ng	InitOperations.java	initLogging	System.err.println() printStackTrace()
2	org.apache.struts2.dispatcher	DefaultStaticContentLoader.java	initLogging	System.err.println() printStackTrace()
3	org.apache.struts2.dispatcher	FilterDispatcher.java	initLogging	System.err.println() printStackTrace()
4	org.apache.struts2.dispatcher	Dispatcher.java	init_CustomConfigurationProviders	ConfigurationException()
5	org.apache.struts2.impl	StrutsObjectFactory.java	buildInterceptor	ConfigurationException()

In Java, exceptional events are handled by the try, catch, and finally statements. These statements contribute to improve the quality of a software system. We have identified 96 methods that contain the try statement. However, only five of these methods contain one try statement with three or four catch statements and no finally statements. These methods are summarized in Table 5. The first method, InitOperations::initLogging, is shown in Figure 3. Because each of the first three methods in Table 5 contains three catch statements having the same structure, they should be maintained consistently. The last two methods have similar structures and throw exceptions to the ConfigurationException() method, but the types of exceptions thrown are slightly different. These structures are informative for engineers maintaining source codes, and students studying exception handling.

3.2 Synchronized statement

Synchronized statements are only used in 11 methods. We checked all source codes to confirm that HttpSession session is synchronized with get, put, remove and check sessions. Figure 4 shows fragments of source codes used to obtain the attribute of a session associated with a given key and place an attribute with a given key (org.apache.struts2.dispatcher.SessionMap class).

4. Code Retrieval Using Vector Space Model

4.1 Structural Metrics as Vector Components

The vector space model [12] is an algebraic model for representing text documents as vectors of identifiers or terms. Given a set of documents D, a document $\boldsymbol{d}_j$ in D is represented as a vector of term weights:

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

where N is the total number of terms in document $\boldsymbol{d}_j$ and $w_{i,j}$ is the weight of the i-th term.

A user query can be similarly converted into a vector $\boldsymbol{q}$:

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

The similarity between document $\boldsymbol{d}_j$ and query $\boldsymbol{q}$ can be computed as the cosine of the angle between the two vectors $\boldsymbol{d}_j$ and $\boldsymbol{q}$ in the N-dimensional space:

```
public void initLogging( HostConfig filterConfig ) {
  String factoryName = filterConfig.getInitParameter("loggerFactory");
  if (factoryName != null) {
    try {
      Class cls = ClassLoaderUtil.loadClass(factoryName, this.getClass());
      LoggerFactory fac = (LoggerFactory) cls.newInstance();
      LoggerFactory.setLoggerFactory(fac);
    } catch ( InstantiationException e ) {
      System.err.println("Unable to instantiate logger factory: " +
             factoryName + ", using default");
      e.printStackTrace();
    } catch ( IllegalAccessException e ) {
      System.err.println("Unable to access logger factory: " +
             factoryName + ", using default");
      e.printStackTrace();
    } catch ( ClassNotFoundException e ) {
      System.err.println("Unable to locate logger factory class: " +
             factoryName + ", using default");
      e.printStackTrace();
    }
  }
}
```

Figure 3. Example of a method retrieved by specific try-catch structures

```
public V get(Object key) {
  if (session == null) {
    return null;
  }
  synchronized (session) {
    return (V) session.getAttribute(key.toString());
  }
}

public V put(K key, V value) {
  synchronized (this) {
    if (session == null) {
      session = request.getSession(true);
    }
  }
  synchronized (session) {
    V oldValue = get(key);
    entries = null;
    session.setAttribute(key.toString(), value);
    return oldValue;
  }
}
```

Figure 4. Fragments of code using Synchronized statement

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

It is natural to assign structural metrics to the elements of a document vector. For example, the getMapping method shown in Figure 2 consists of seven if statements, one else statement, tree else-if statements and two for statements. Thus, the getMapping method is represented by the vector (0, 0, 0, 0, 7, 1, 3, 2, 0, 0). Each element of the vector corresponds to a "synchronized," "try," "catch," etc. statement as shown in Table 1.

Although this idea is appealing, there is an essential defect. For example, the similarity of vectors (0, 0, 0, 0, 7, 1, 3, 2, 0, 0) and (0, 0, 0, 0, 14, 2, 6, 4, 0, 0) is 1.0, because the two vector have the same direction. However, source codes composed of 7 if statements are obviously different from those composed of 14 if statements. This defect is often observed in vectors that contain only a few elements with non-zero values.

The Euclidean distance between vectors $\boldsymbol{q}$ and $\boldsymbol{d}$ is the distance of the vector $|\ \boldsymbol{v}\ | = |\ \boldsymbol{q} - \boldsymbol{d}\ |$. In general, for an N-dimensional space, the distance is defined by the magnitude of the vectors and is computed in component form by the following formula:

$$|\boldsymbol{v}| = |\ \boldsymbol{q} - \boldsymbol{d}\ | = \sqrt{\sum_{i=1}^{n}(q_i - d_i)^2}$$

The magnitude, termed squared Euclidean distance, is frequently used in various disciplines when the magnitude of differences has to be compared. We use the distance as a difference measure.

Figure 5 illustrates the concept of similarity and difference in the context of vector algebra. Intuitively, while similarity depends on the directions of two vectors, the difference depends on the length of the vector resulting from the subtraction of the two vectors. Because vectors represent simultaneously both magnitude and direction, the similarity and difference measures naturally characterize the vectors under consideration.

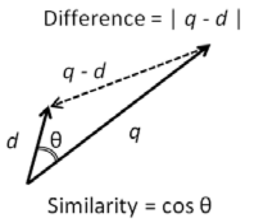

Figure 5. Concept of similarity and difference measures

4.2 Results of Code Retrieval

Table 6 shows the top ten methods obtained by retrieving source code that includes one try statement and one final statement. The methods in Table 6 are sorted first by difference and then similarity. By assigning higher priority to the difference, only meaningful records are listed at the top positions. The top four methods are comprised of almost identical code segments, as shown in Figures 6, and 7.

The three init methods and the contextInitialized method throw different handling exceptions, i.e., the init methods throw ServletException (Figure 6), while contextInitialized does not (Figure 7). The resulting exception triggers the investigation of the code in more detail.

Table 6 also indicates other methods containing the same code segments. In fact, the contents of the two end methods of Submit.java and UIBean.java consist of almost the same sequence of statements, as shown in Figures 8 and 9, respectively. However, they differ in how they handle exceptions, i.e., the former writes an error message in a log file, while the latter throws an exception to StrutsException() that is implemented in Strut 2 Core.

```
public void init(ServletConfig filterConfig) throws ServletException {
   InitOperations init = new InitOperations();
   try {
      ServletHostConfig config = new ServletHostConfig(filterConfig);
      init.initLogging(config);
      Dispatcher dispatcher = init.initDispatcher(config);
      init.initStaticContentLoader(config, dispatcher);

      prepare = new PrepareOperations(filterConfig.getServletContext(), dispatcher);
      execute = new ExecuteOperations(filterConfig.getServletContext(), dispatcher);
   } finally {
      init.cleanup();
   }
}
```

Figure 6. Init method in StrutsServlet.java

Table 6. Methods including a try-final-statement obtained by source code retrieval

No	Package Name	File / Class Name	Method Name	Similarity	Difference	Code Lines
1	org.apache.struts2.dispatcher.ng.servlet	StrutsServlet.java	init	1.0000	0.0000	13
2	org.apache.struts2.dispatcher.ng.listener	StrutsListener.java	contextInitialized	1.0000	0.0000	13
3	org.apache.struts3.dispatcher.ng.filter	StrutsPrepareFilter.java	init	1.0000	0.0000	13
4	org.apache.struts2.dispatcher.ng.filter	StrutsPrepareAndExecuteFilter.java	init	1.0000	0.0000	15
5	org.apache.struts2.dispatcher	FilterDispatcher.java	init	1.0000	0.0000	12
6	org.apache.struts2.interceptor	StrutsConversionErrorInterceptor.java	getOverrideExpr	1.0000	0.0000	9
7	org.apache.struts2.util	MergeIteratorFilter.java	next	1.0000	0.0000	7
8	org.apache.struts2.components	Submit.java	end	0.8165	1.0000	14
9	org.apache.struts2.components	UIBean.java	end	0.8165	1.0000	13
10	org.apache.struts2.components	Component.java	copyParams	0.8165	1.0000	13


```
public void contextInitialized(ServletContextEvent sce) {
  InitOperations init = new InitOperations();
   try {
      ListenerHostConfig config = new ListenerHostConfig(sce.getServletContext());
      init.initLogging(config);
      Dispatcher dispatcher = init.initDispatcher(config);
      init.initStaticContentLoader(config, dispatcher);

      prepare = new PrepareOperations(config.getServletContext(), dispatcher);
      sce.getServletContext().setAttribute(StrutsStatics.SERVLET_DISPATCHER, dispatcher);
   } finally {
      init.cleanup();
   }
}
```

Figure 7. ContextInitialized method in StrutsListener.java

```
public abstract class UIBean extends Component {
   public boolean end(Writer writer, String body) {
      evaluateParams();
      try {
         super.end(writer, body, false);
         mergeTemplate(writer,
                  buildTemplateName(template, getDefaultTemplate()));
      } catch (Exception e) {
         throw new StrutsException(e);
      }
      finally {
         popComponentStack();
      }
      return false;
   }
```

Figure 8. The end method in Submit.java

```
public class Submit extends FormButton {
   public boolean end(Writer writer, String body) {
      evaluateParams();
      try {
         addParameter("body", body);
         mergeTemplate(writer,
                  buildTemplateName(template, getDefaultTemplate()));
      } catch (Exception e) {
         LOG.error("error when rendering", e);
      }
      finally {
         popComponentStack();
      }
      return false;
   }
```

Figure 9. The end method in UIBean.java

Although syntax matching of control structures is employed, our approach retrieves similar source code using a characteristic structure as a query. Table 7 shows the top twelve methods obtained by using a query vector with the components (0, 0, 0, 0, 11, 1, 0, 0, 0, 0), i.e., retrieving source code that includes eleven if-statements and one else-statement.

```
public class UpDownSelect extends Select {
  ...
  public void evaluateParams() {
    super.evaluateParams();
    // override Select's default
    if (size == null || size.trim().length() <=0) {
      addParameter("size", "5");
    }
    if (multiple == null || multiple.trim().length() <=0) {
      addParameter("multiple", Boolean.TRUE);
    }
    if (allowMoveUp != null) {
      addParameter("allowMoveUp", findValue(allowMoveUp, Boolean.class));
    }
    if (allowMoveDown != null) {
      addParameter("allowMoveDown", findValue(allowMoveDown, Boolean.class));
    }
    if (allowSelectAll != null) {
      addParameter("allowSelectAll", findValue(allowSelectAll, Boolean.class));
    }
    if (moveUpLabel != null) {
      addParameter("moveUpLabel", findString(moveUpLabel));
    }
    if (moveDownLabel != null) {
      addParameter("moveDownLabel", findString(moveDownLabel));
    }
    if (selectAllLabel != null) {
      addParameter("selectAllLabel", findString(selectAllLabel));
    }
    // inform our form ancestor about this UpDownSelect so the form knows how to
    // auto select all options upon it submission
    Form ancestorForm = (Form) findAncestor(Form.class);
    if (ancestorForm != null) {
      // inform form ancestor that we are using a custom onsubmit
      enableAncestorFormCustomOnsubmit();
      Map m = (Map) ancestorForm.getParameters().get("updownselectIds");
      if (m == null) {
        // map with key -> id ,  value -> headerKey
        m = new LinkedHashMap();
      }
      m.put(getParameters().get("id"), getParameters().get("headerKey"));
      ancestorForm.getParameters().put("updownselectIds", m);
    }
    else {
      if (LOG.isWarnEnabled()) {
           LOG.warn("no ancestor form found for updownselect "+this+",
           therefore autoselect of all elements upon form submission will not work ");
      }
    }
  }
```

Figure 10. The evaluateParams method in UpDownSelect.java

Table 7. Methods including eleven if-statements and one else-statement obtained by source code retrieval

No	Package Name	File / Class Name	Method Name	Similarity	Difference	Code Lines
1	org.apache.struts2.components	UpDownSelect.java	evaluateParams	1.0000	0.0000	41
2	org.apache.struts2.components	InputTransferSelect.java	evaluateExtraParams	1.0000	0.0000	50
3	org.apache.struts2.components	Form.java	evaluateExtraParams	0.9959	1.0000	36
4	org.apache.struts2.views.freemarker	ScopesHashModel.java	get	0.9959	1.0000	40
5	org.apache.struts2.dispatcher.mapper	DefaultActionMapper.java	getUriFromActionMapping	0.9959	1.4142	43
6	org.apache.struts2.components	ComboBox.java	evaluateExtraParams	0.9938	2.2361	39
7	org.apache.struts2.interceptor	ServletConfigInterceptor.java	intercept	0.9959	2.2361	35
8	org.apache.struts2.interceptor	FileUploadInterceptor.java	intercept	0.9840	3.1623	67
9	org.apache.struts2.interceptor	ExecuteAndWaitInterceptor.java	doIntercept	0.9739	3.4641	57
10	org.apache.struts2.interceptor	FileUploadInterceptor.java	getTextMessage	0.9759	3.4641	60
11	org.apache.struts2.interceptor	FileUploadInterceptor.java	acceptFile	0.9491	3.6056	39
12	org.apache.struts2.components	File.java	evaluateParams	0.9959	4.1231	24

```
protected void evaluateExtraParams() {
    super.evaluateExtraParams();
    if (validate != null) {
        addParameter("validate", findValue(validate, Boolean.class));
    }

    if (name == null) {
        //make the name the same as the id
        String id = (String) getParameters().get("id");
         if (StringUtils.isNotEmpty(id)) {
           addParameter("name", id);
         }
    }

    if (onsubmit != null) {
        addParameter("onsubmit", findString(onsubmit));
    }

    if (onreset != null) {
        addParameter("onreset", findString(onreset));

    }
    if (target != null) {
        addParameter("target", findString(target));
    }

    if (enctype != null) {
        addParameter("enctype", findString(enctype));
    }

    if (method != null) {
        addParameter("method", findString(method));
    }

    if (acceptcharset != null) {
        addParameter("acceptcharset", findString(acceptcharset));
    }

    // keep a collection of the tag names for anything special the templates
    //might want to do (such as pure clientside validation)
    if (!parameters.containsKey("tagNames")) {
        // we have this if check so we don't do this twice (on open and close of the template)
        addParameter("tagNames", new ArrayList());
    }
    if (focusElement != null) {
        addParameter("focusElement", findString(focusElement));
    }
}
```

Figure 11. The evaluateExtraParams method in Form.java

The evaluateParams method in UpDownSelect.java file is shown in Figure 10, and the evaluateExtraParams method in Form.java file is shown in Figure 11.

The evaluateParams method performs first super.evaluateParams method for populating parameters, and then addParameter method for maintaining a parameter list with respect to each value of parameters. The other three methods of No. 2, 6, and 12 in Table 7 consist of approximately the same control statements. The five metods in the org.apache.struts2.interceptor packge are implemented in a similar manner including usage of proprietary method in the package.

The results are of benefit to engineers and students to study cording techniques in a given context. Because this approach only uses source codes, in case technical documents are lost, the results of retrieval provide effective measures for maintenance engineer to collect source codes that should be considered in a consistent manner.

5. Conclusions and Future Work

Open-source programs represent a tremendous resource of exceptional code that could be used not only for educational purposes but also for developing practical Web applications. However, due to the vast amounts of available source codes, it is difficult to find efficiently the code segments that we want. Information retrieval techniques can help us extract potential coding knowledge from source codes.

In this paper, we presented an approach that improves the retrieval of source code using structural information of control statements. We have conducted two types of experiments. In the first, we retrieved the code using the characteristic structure as a key. In the second we used a vector space model in which structural metrics were assigned to each element of a vector. In both experiments, our methods retrieved several sets of source codes that are presumably maintained in a consistent manner.

A key contribution of our approach is the incorporation of a difference measurement that improves the vector space model. The difference measurement was proven especially effective in distinguishing vectors that have the same direction but differ in length.

The results are promising enough to warrant future research. In this study, we focused only on structures of control statements, and mapped them into a document vector in the vector space model. In future work, we will work on improving our methods by combining semantic information, such as instantiation of a class and implementation of an abstract class, etc. into structural information. We will also conduct experiments on various types of open source programs available on the Internet.

References

[1] Android open source project. 2012. http://source.android.com/.

[2] Baker, B.S. 1996. Parameterized pattern matching: algorithms and applications. Journal Computer System Science 52, 1 (February 1996), 28-42.

[3] Baxter, I. D., Yahin, A., Moura, L. Sant'Anna, M., and Bier, L. 1998. Clone detection using abstract syntax trees. Proc. of the 14th International Conference on Software Maintenance (November 1998), pp.368-377.

[4] Horwitz, S., Reps, T. W., and Binkley, D. 1990. Interprocedural slicing using dependence graphs, ACM Trans. on Programming Lang. and Sys. 12, 1 (January 1990), pp.1-34.

[5] Jiang, L., Misherghi, G., Su, Z., and Glondu, S. 2007. DECKARD: Scalable and accurate tree-based detection of code clones, Proc. of the 29th international conference on Software Engineering (May 2007), 96-105.

[6] Kamiya, T., Kusumoto, S., and Inoue, K. 2002. CCFinder: A multi-linguistic tokenbased code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28, 7 (July 2002), 654-670.

[7] Di Lucca, G., Di Penta, M., Fasolino, A. 2002. An approach to identify duplicated web pages. Proc. of the 26th international Computer Software and Applications Conference (August 2002), 481-486.

[8] Marcus, A., and Maletic, J. 2001. Identification of high-level concept clones in source code. Proc. of the 16th international Conference on Automated Software Engineering (November 2001), 107-114.
[9] Mayland, J., Leblanc, C., and Merlo, E.M. 1996. Experiment on the automatic detection of function clones in a software system using metrics, Proc. of the 12th International Conference on Software Maintenance (November 1996), 244-253.
[10] McCabe, T.J. 1976. A complexity measure, IEEE Transactions on software engineering, 2, 4 (December 1976), 308-320.
[11] McCreight, E. 1976. A space-economical suffix tree construction algorithm. Journal of the ACM 23, 2 (April 1976), 262-272.
[12] Salton, G. and Buckley, C., 1988. Term-Weighting approaches in automatic text retrieval, Information Processing and Management, 24, 5 (November 1988), 513-523.
[13] SourceForge. 2012. http://sourceforge.net/.
[14] Struts - The Apache Software Foundation. 2012. http://struts.apache.org/.