VNU Journal of Science: Comp. Science & Com. Eng., Vol. 32, No. 3 (2018) 1–10

A Toolchain for Source Code Quality Assurance of Java EE Applications

Cuong Quang Bui[*], Loc Tien Dinh, Anh Viet Luu, Hoa Viet Nguyen, Quy Ngoc Pham, Pham Ngoc Hung

Faculty of Information Technology,
VNU University of Engineering and Technology,
144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

Abstract

This paper presents a novel tool named JCIA-VT for source code analysis and quality assurance of Java EE applications using Hibernate, Struts, and Spring frameworks. Specifically, the tool applies static program analysis to construct a data structure called Java Dependency Graph (JDG) demonstrating the dependencies among components of a Java EE project, then uses this structure to predict impacted components under a set of changed components. Besides, JDG is used as an input for analyzing several structural analysis functions including business data graph, and cyclomatic complexity computation. The experimental results have been shown the effectiveness of the tool in practice for both developers and testers to analyze change impact and understand application systems.

Received May 2018, Revised May 2018

Keywords: program analysis, change impact analysis, data flow, enterprise application

1. Introduction

Nowadays, the enterprise applications are usually developed in a long time with high complexity. After many update patches, they don't have complete specification and design documents. Even the source code becomes the only document in many cases. Meanwhile, maintenance and upgrade happen regularly. To ensure software quality for the new version, the development team need to test the entire system. This is impossible because of enormous cost. As a result, the development team can not control full impact of the changes, which may lead to significant risks for the business during the operation. This is a difficult problem because the solutions depend closely on the used technology. Therefore, proposing the solutions and developing appropriate tools to address this problem is one of big challenges and received research attention.

Change Impact Analysis (CIA) is considered as a solution to solve the problem. CIA plays an important role in the stages of

[*] Corresponding author. Email: 14020577@vnu.edu.vn

development, maintenance and regression testing. CIA can be an impact assessing and cost estimating tool. It also helps to reduce the number of required test cases. The CIA technique has two main approaches, including static CIA and dynamic CIA [1]. In fact, results of researches indicate the methods related to static CIA such as CIA Based on Change Types [2], Control Call Graph-based Technique [3], and WAVE-CIA [4]. The inputs of CIA techniques are the result of dependency analysis from the application. This process cannot be generalized to all available technologies and platforms.

Java EE is a popular solution to deploy modern enterprise web applications with many core technologies and many other frameworks. To solve this problem for Java EE applications, a tool called JCIA [5] has been proposed and developed. However, this method is still rudimentary and can only analyze some built-in technologies of Java EE such as JSF, CDI, JAX-WS. JCIA is unfinished and unattainable high efficiency. This tool cannot analyze completely a practical Java EE project. Industrial projects usually are built by a complex bundle of technologies and frameworks rather than Java EE built-in technologies.

Therefore, a complete method has been researched and proposed to perform change impact analysis for cross-platform Java EE applications. Popular frameworks such as Spring, Struts, Hibernate will be supported at first. Then we will gradually improve the method for all other platforms. In addition, we also propose other solutions to provide the additional objective views of the application system such as building data flow, visualizing architecture and computing cyclomatic complexity. A toolchain named JCIA-VT has been developed to implement the proposed solution.

The rest of this paper is structured as follows. Section 2 presents the methods of implementation including preprocessing source code; dependency analysis for Java Core, Struts, database technology. Section 3 and Section 4 describe some approaches to analyze change impact and building business data flow. Next, Section 5 presents the JCIA-VT toolchain and results are obtained through experiment analysis. Finally, Section 6 summarizes the obtained results, conclusions, drawbacks and directions for research and development in the future.

2. Dependency Graph Generation

Definition 1. *(Java EE Dependency Graph). Given a Java EE project, a Java EE Dependency Graph, denoted JDG, is defined as a pair* (V, E)*, where* $V = \{v_0, v_1, ..., v_n\}$ *is a list of nodes representing components such as packages, files, classes, methods, attributes, etc. and* $E = \{(v_i, v_j) | v_i, v_j \in V \subseteq V \times V\}$*, is a list of directed edges. Each edge* (v_i, v_j) *represents a dependency between* v_i *and* v_j *that means* v_i *depends on* v_j*.*

Before building the JDG, the given Java EE project had been preprocessed to construct a structure tree whose nodes represent project components such as folders, files, classes, methods, attributes, XML tags, etc. First, every file of the project is scanned and added to the tree. Then, the checker analyzes and defines what type of these files and their children are. A Java EE project usually contains Java and

XML-based source code. If a file is a Java source code file, its children are classes, methods, and attributes. Those children are XML-tags if that file is an XML-based source file. Finally, this tree would be used to construct a JDG of which vertices are nodes of the tree and edges are dependencies between source code components. These dependencies are collected and added to JDG by dependency analyzers corresponding to Java EE technologies and frameworks such as Struts, Hibernate, JDBC.

2.1. Collecting Java Core dependencies

There are three major types of dependencies in Java Core: inheritance, method invocation, and field access. In the Change Impact Analyzer, three independent components are corresponding to each of the dependencies mechanisms. For any given Java source files, the Abstract Syntax Tree (AST) would be generated by using Java Development Tool (JDT), then the dependencies among the components are identified and inserted into JDG based on the obtained AST.

For instance, from given sample source code in Fig 1, Java Core dependency analyzer could detect three types of dependencies: inheritance from node *A* to node *B*, field access from node *A/f()* to node *B/f()* and method invocation from node *B/f()* to node *A/f()*.

2.2. Collecting Struts dependencies

Struts is a popular open-source Java EE framework which is built on top of Servlet API and JavaServer Pages. It is designed based on Model-View-Controller (MVC) design pattern and allows developers to define clearly Controller, Model and View in order to build maintainable and understanding applications. Dependency relationships of Struts usually exist between source code elements corresponding to components in MVC.

```
public class A {
  int a;
  public int f() {return a++;}
}
class B extends A {
  public int f() {return super.f()+5;}
}
```

Figure 1. Example of Java program

Struts configuration components written in XML format need collecting at first because they determine the definitions of Struts components (*Package*, *Action*, *Result*, *Interceptor*, etc) in the application which provide necessary information to define dependencies among them. Because the definitions of components could be written discretely in multiple files (e.g. configural information of a *Package* can be declared in separate files, Struts engine collects them when the application is running), it is hard to collect configuration information and define dependencies at the same time. To solve this problem, a refinement process would be performed before defining dependencies to aggregate Struts configurations. The new XML-file vertices contain additional configuration components information are created and replaced for the old ones on JDG even though they still keep necessary information by using *Decorator* design pattern.

The new vertices are analyzed to define

```
<action name="AddBookAction"
class="vn.sample.action.AddBookAction">
  <result name="success">
    /addbook-success.jsp
  </result>
  <result name="error">
    /addbook-error.jsp
  </result>
</action>
```

Figure 2. Example of Struts configural source

```
<html>
  <h1>You've added book
    <s:property value=''bookName''>
  </h1>
</html>
```

Figure 3. Example of data usage in Struts

dependencies. Almost type of dependencies can be retrieved directly by analyzing Struts definitions. They are dependencies from Controller to View and Model components, some of them are dependencies between configural components. Figure 2 shows an example of Struts configural source. After refining process and analyzing this source, three dependencies are easily defined: *ActionConfigurationDependency* from `<action>` tag to `AddBookAction` class, the rest are *ResultConfigurationDependency* from `<result>` tags to `addbook-error.jsp` and `addbook-success.jsp`. There are also dependencies from View to Model, they represent for data usage relationships to Model components. Configural information from refining process is necessary to define these components. Figure 3 shows the content of `addbook-error.jsp`. `<s:property>` is

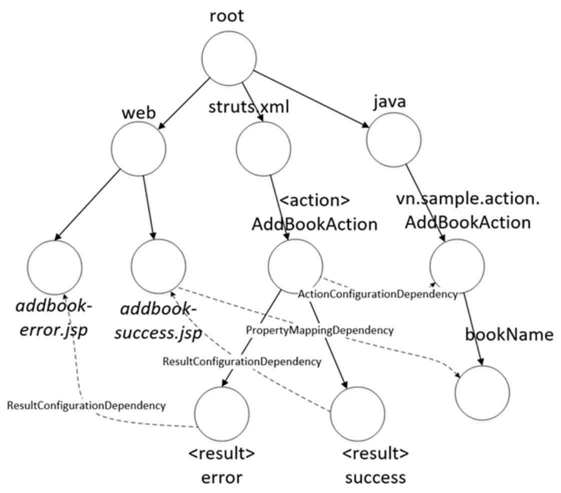
root
web
struts.xml
java
<action> AddBookAction
vn.sample.action. AddBookAction
addbook-error.jsp
addbook-success.jsp
bookName
ActionConfigurationDependency
PropertyMappingDependency
ResultConfigurationDependency
ResultConfigurationDependency
<result> error
<result> success

Figure 4. Flow graph convertion rules

a Struts tag for accessing data from Model. Data property has the name `bookName`. To find out where `bookName` is declared, we have use configural definitions to check which action mapping with this View. Here, that is `AddBookAction` and the Model is `AddBookAction` class. Therefore, we know that `bookName` comes from that class. A *PropertyMappingDependency* is created, starts from `<s:property>` tag of `addbook-success.jsp` to `bookName` field of `AddBookAction` class. The result of Struts depenency analyzer is shown in Fig 4.

2.3. *Collecting database dependencies*

Java applications use JDBC for connecting to database and executing SQL queries to retrieve data. The aim of database dependency analysis is to try to recreate queries from source code and determine the affected tables. The method `getStock()` in Fig 5 illustrates how SQL query is dynamically generated for a JDBC binding to a database. The query analysis starts by finding all locations using

```
ResultSet getStock(int id, int ord) {
  String q = "SELECT * FROM stock"
        + " WHERE stockId = " + id;
  if (ord == -1)
    q = q + " ORDER BY id ASC";
  else if (ord == 1)
    q = q + " ORDER BY id DESC;
  return session.executeQuery(q);
}
```

Figure 5. Example of a method using SQL Query

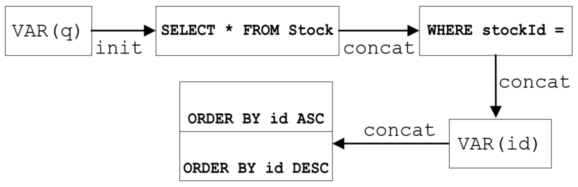

Figure 6. String flow graph of `getStock` method

method `executeQuery` in source code, called *hotspot*. Next, the analysis creates a *flow graph* representing the creation of all possible string expressions. The vertices in a flow graph corresponding to variables, methods, expressions or operations (`replace()`, `insert()`, `trim()`, ...). If string expressions is used in condition statement, such as `switch-case` or `if-else`, the flow graph will put expressions of all clauses in one vertex. The edges use two label "*init*", "*concat*" to describe how the next vertices is used by the previous vertex. The label "*init*" is only used by variable vertex or method vertex. The flow graph for `getStock` method looks as Fig 6. In the next step, the flow graph will be converted to string value. First, variable and method vertices which have "*init*" edge are removed. Flow graph remains only "*concat*" edges. Then we construct a

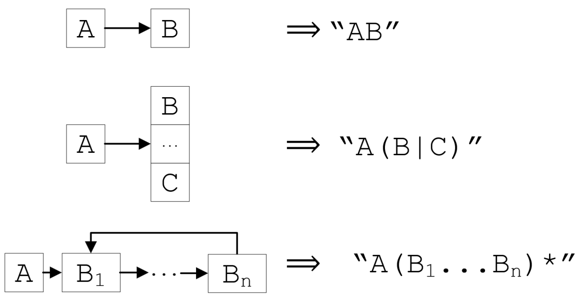

Figure 7. Flow graph conversion rules

set of rules for the string generation. Detail of these rules is described in Fig 7. The result of conversion is a string value in regular expression form. For the `getStock` method, the generated string value is `SELECT * FROM stock WHERE stockId = id (ORDER BY id ASC|ORDER BY id DESC)`. Because `id` is an external variable, the convertion does nothing with this variable and keep its name as a part of result string. After completely generating the string value, the analysis splits out each word of query, finds the name of tables and generates dependencies between the invoking methods and used tables.

3. Change Impact Analysis

The change impact analysis could be performed based on two major approaches: static information analysis or dynamic information analysis. The static information analysis contains three main methods such as structural static analysis, textual analysis, historical analysis. Currently, the researches on CIA are mainly focused on structural analysis which consists of two primary phases, generating dependency graphs of applications, and calculating potential change impact sets based on those graphs. There are several algorithms to calculate the impact sets at method or attribute level and combine CIA based on change types and WAVE-CIA which is used in the change impact analyzer has been known as an appropriate solution to solve that problem.

The main idea of WAVE-CIA method is the water-wave propagation [4]. The ripples on water surface will meet as they spread and generate new spreading centers. Like this natural process, the WAVE-CIA procedure follows two steps: identifying the core set from the Change Set (CS) of the call graph and computing Potential Impact Set (PIS) by adding more components generated from the propagation analysis of the core. In the toolchain, this new CIA approach is proposed to be implemented automatically by giving the change set as the result of comparing the source code between two versions.

4. Data Flow Analysis

A data flow graph (DFG) represents a task or a real-life business function of an application. It contains an ordered set of application's components, describes the movement of data in the application and how components transfer data to each other. To analyze data flow of applications, we consider two type of dependency: *data dependency* and *logical dependency*. The logical dependency is created by the relations of two nodes such as is-a, has-a or parent-child and the data dependency is created by data usage in two nodes. After determining the types of dependency, we define a function that transforms these dependencies into a list of vertices and edges of DFG. Figure

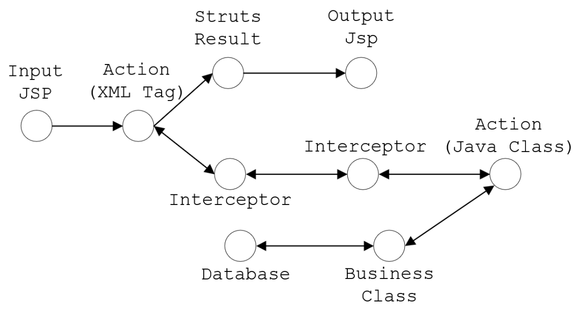

Figure 8. Example of data flow generated by converting data dependency

8 illustrates a typical data flow of Struts application. The first vertex of the graph, which is usually a JSP file, starts the flow of data. Each `<form>` tag is mapped to a *Struts Action* corresponding to a Java class which implemented the application's logic and retrieve data from the database. Struts Action also consists of a list of *Struts Result* which define the JSP file that displays final data. Because the created graph is not expressed the sequence of data transfer, the graph needs normalizing. We define two types of edge in data flow graph, *Request Edge* and *Response Edge*. Request edges represent for data transferred from View layer to Database layer, and response edges represent for data with the inverse direction. The type of each edge is specified in transformation function of data dependency.

To facilitate the normalization and visualization, the graph has to be converted to a simple form. All edges are still preserved in the simple graph but vertices only remain necessary attributes from the origins for displaying such as id, name, path, type. The normalization of data flow follows two steps. The first step is eliminating all bi-direction edges. A bi-direction edge exists when two

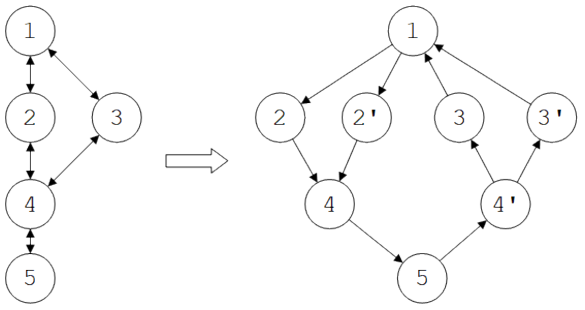

Figure 9. Eliminate bi-direction step in data flow graph normalization

vertices have both request and response edges between them. The purpose of this step is to make sure all incoming and outgoing edges of a vertex have only one type. However, there are some special vertices called *Endpoint* include both types of edge (e.g. the database vertice). The original vertex keeps one type of edge and all edges of the other types are moved to the new vertex. Figure 9 explains the elimination. From endpoint vertex (fifth vertex), the graph will be traversed up to find all vertices contain bi-direction edges (vertex 2, 3, 4). The original vertices retain only request edges. All the response edges are moved to vertices 2', 3', 4' which are the duplication of vertices 2, 3, 4. The second step of the normalization is cycle removing. A cycle exists if it still has both request edge and response edge after bi-direction elimination step. A graph containing cycles could not show the start and end vertices using the data. The solution to remove cycle is similar to bi-direction elimination step. Figure 10 clarified the removal.

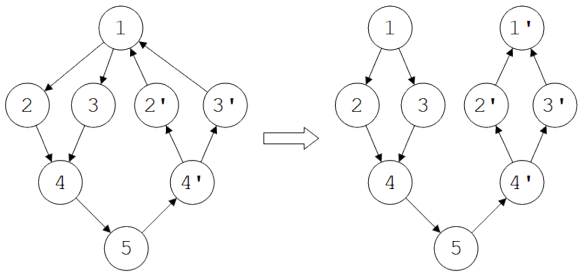

Figure 10. Cycle removal step in data flow graph normalization

5. Tool and Experiments

5.1. Tool Implementation

JCIA-VT has been implemented in Java and can be deployed as a Web application with architecture in Fig 11. Given source code of a Java EE project, the Preprocessor module analyses the whole project to construct Java dependency graph of source code components. Next, Dependency Analyzer module including four dependency analyzers corresponding to Java EE technologies and frameworks determine dependency relationships between components. This graph is used as input for Change Impact Analyzer and Structural Analyzer modules. The result of analyzing is shown to users by Visualizer module which has been implemented by D3.js[1].

JCIA-VT is a completely upgrade version of JCIA [5], added structure analysis and a more efficiently change impact analyzer method. JCIA-VT would display the JDG of application's source code and let users choose the changed components. However, the chosen components are marked as blue and the impacted ones are red. This change makes users differentiate analyzed nodes of

[1]https://d3js.org

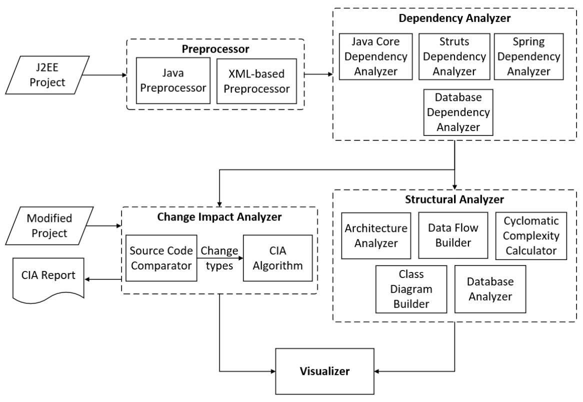

Figure 11. Architecture of JCIA-VT

JDG with their types. Figure 12 shows a *Dependency View* of a JCIA-VT. Moreover, JCIA-VT provides users to choose depth level of the change impact analysis. This feature allows analyzer restrict the number of impacted components for purposes of users.

Beside the dependency view and change impact implementation, JCIA-VT also provides three structure views:

- *Data Flow View*: analyze and show a data flow graph based on the data movement between components.
- *Class Diagram*: describe the structure of application by showing classes, their attributes, methods and relationship among objects. JCIA-VT's class diagram obeys the UML 2.5.1 spec[2].
- *Cyclomatic*: calculate the complexity of classes based on source code using many criteria. Each class is represented as a cycle. The size and color of cycle depend on LOC (Line Of Code) per class and the source code's complexity.

[2]https://www.omg.org/spec/UML/2.5/PDF

Table 1. Experiments on case studies of JCIA-VT

	Sample 1	**Sample 2**
# of files	1,177	1,739
LOC	235,969	363,021
# of vertices	43,667	106,556
# of dependencies	36,961	88,281
Time	38s	69s
Memory	760MB	1300MB

5.2. *Experiments*

In order to show the effectiveness of JCIA-VT, we tested by using Java EE projects which are provided by Viettel Technology and Software Quality Management Center. They both use Struts, Hibernate and JDBC technology. Table 1 shows experiments when JCIA-VT analyzes case study projects on a laptop with Intel Core i5-3320M CPU @ 2.6GHz and 8GB Memory.

Result of change impact analysis using WAVE-CIA with *depth = 1* is shown in Fig 12. Methods *editProfileConfig()* and *addProfileConfig()* are requested to change. CIA algorithm computes Core Set such as fields *log*, *profileConfigForm*; methods *copyBeanFromBO()*, *checkExistActionCode()*; and database *PROFILE_CONFIG* because both two elements in change set have dependency relationships with them. Next, PIS is computed from Core Set. That is method *findById()* because it calls elements in Core Set and an element in Core Set also calls it. As a result, the final impacted elements are overlaid with red color in Fig 12.

Figure 13 represents a data flow graph generated by using the *Sample 2* project. The flow graph is started from `prepareChannelType.jsp`. The data of this JSP file is transfered to a Struts

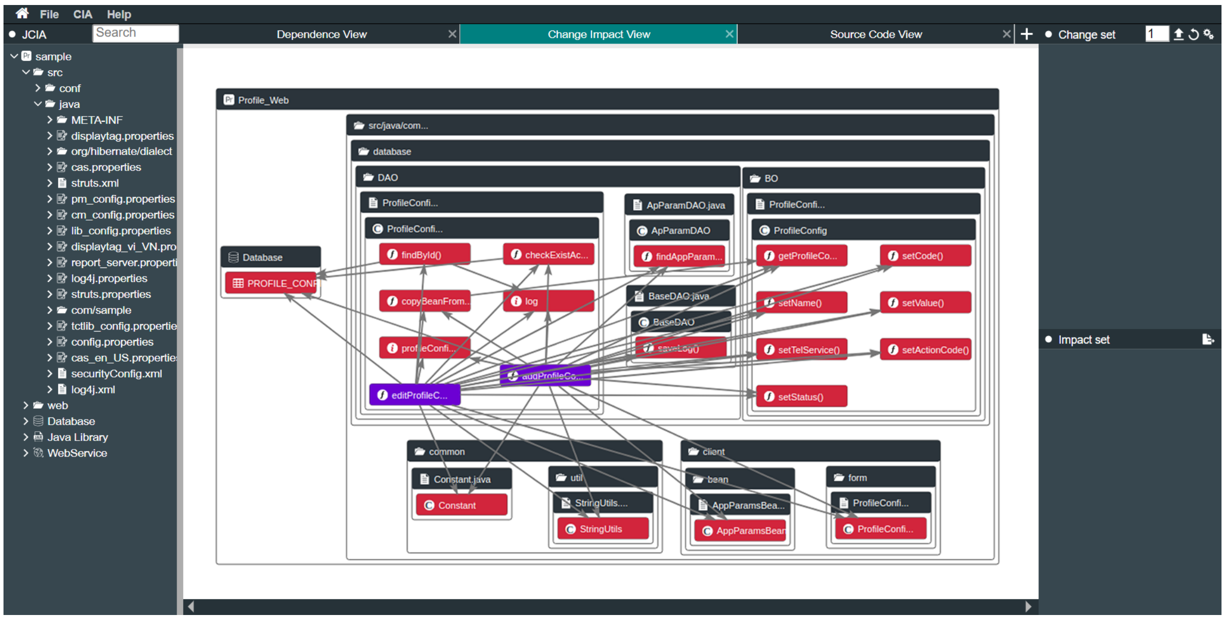

Figure 12. User Interface of JCIA-VT

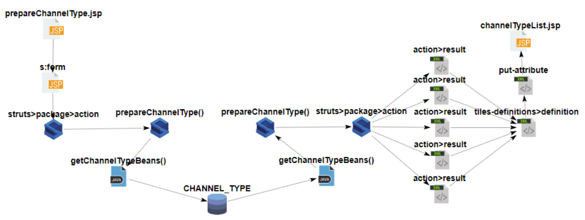

Figure 13. Example of Data flow analyzed by JCIA-VT

Action tag node (<action> tag). Moreover, this Action node is mapped to an Action Method, `prepareChannelType()`. We also detect the method `getChannelTypeBean()` which retrieves data from `CHANNEL_TYPE` table in database, is used in `prepareChannelType()`. After querying the database, `getChannelTypeBean()` returns all data to `prepareChannelType()` and this Action Method continues doing its business. Action tag node `<action>` defines five `<result>` children tags that match up all possible cases of return data in `prepareChannelType()`. All these result tags both use the same `<tile-definition>` which declares `channelTypeList.jsp` in its `put-attribute` property. All the movement of data we just genenerated can be considered a real life business. Based on the data, users fill in the form in `prepareChannelType.jsp`, JCIA-VT will show all the data (in this case is the channels) from database to `channelTypeList.jsp` view.

6. Conclusion

This paper has presented a tool and proposed several methods of source code quality assurance of Java EE applications. At this time, the main functions of this toolchain have been basically completed in order to analyze the source code of applications which were built based on Struts 2 and Hibernate framework. This toolchain provides not only the change impact analysis at method or attribute level in Java source code files but also business data flow analysis of applications based on constructing JDG and considering different types of dependencies. Besides, it is also integrated a graphical user interface extended from D3 Javascript library to visualize the results of these analyses and allow users to interact with the toolchain.

In the future, we plan to upgrade this toolchain to increase the correctness of change impact set by including more dependencies on components or applying more complicated and accuracy CIA technologies. First, the version comparison management toolchain would be integrated with others common version control tools like Git[3] and SVN[4] which would let the change impact analysis and result reporting could be done automatically. Next, an application source code components to views and functions mapping should be created to make this toolchain truly effective and be used broadly in quality assurance procedures in businesses. Furthermore, we would also conduct researches on taking more perspectives of applications so that users could add different information about their applications such as sequence diagram between packages, control flow diagram, assessment tools of source code complexity and application security, etc. Finally, we are exploring more to enhance this toolchain to make it support not only more Java basis and technology but also others programming languages like C#, PHP, JavaScript, etc.

[3]https://git-scm.com

[4]https://subversion.apache.org

Acknowledgment

We would thank Dr. Vo Dinh Hieu, VNU University of Engineering and Technology for his useful reviews.

References

[1] B. Li, X. Sun, H. Leung, S. Zhang, A survey of code-based change impact analysis techniques 23.

[2] X. Sun, B. Li, C. Tao, W. Wen, S. Zhang, Change impact analysis based on a taxonomy of change types, in: Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference, COMPSAC '10, IEEE Computer Society, 2010, pp. 373–382.

[3] L. Badri, M. Badri, D. St-Yves, Supporting predictive change impact analysis: A control call graph based technique, in: Proceedings of the 12th Asia-Pacific Software Engineering Conference, APSEC '05, IEEE Computer Society, 2005, pp. 167–175.

[4] B. Li, Q. Zhang, X. Sun, H. Leung, Wave-cia: A novel cia approach based on call graph mining, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, ACM, 2013, pp. 1000–1005.

[5] L. Ba Cuong, V. Son Nguyen, N. Duc Anh, P. Ngoc Hung, D. Hieu Vo, Jcia: A tool for change impact analysis of java ee applications, 2018, pp. 105–114.