Automatic Extraction of Dependencies between Web Components and Database Resources in Java Web Applications

Jaewon Oh\textsuperscript{1}, Woo Hyun Ahn\textsuperscript{2*}, and Taegong Kim\textsuperscript{3}, Member, KIICE

\textsuperscript{1}School of Computer Science and Information Engineering, The Catholic University of Korea, Bucheon-si 14662, Korea
\textsuperscript{2}School of Software, Kwangwoon University, Seoul 01897, Korea
\textsuperscript{3}Department of Computer Engineering, Inje University, Gimhae 50834, Korea

Abstract

Web applications typically interact with databases. Therefore, it is very crucial to understand which web components access which database resources when maintaining web apps. Existing research identifies interactions between Java web components, such as JavaServer Pages and servlets but does not extract dependencies between the web components and database resources, such as tables and attributes. This paper proposes a dynamic analysis of Java web apps, which extracts such dependencies from a Java web app and represents them as a graph. The key responsibility of our analysis method is to identify when web components access database resources. To fulfill this responsibility, our method dynamically observes the database-related objects provided in the Java standard library using the proxy pattern, which can be applied to control access to a desired object. This study also experiments with open source web apps to verify the feasibility of the proposed method.

Index Terms: Database resource, Dependency relation, Page generation graph, Web component, Web engineering

I. INTRODUCTION

As noted in previous research [1], web applications have become increasingly complex [2, 3] and are rapidly changing [4]. However, documentation on these apps has not been done well enough [3, 5]. Therefore, web app developers encounter difficulties maintaining apps [2, 6, 7], and there is insufficient research to understand them [1, 7]. In addition, Java web apps are considered representative legacy apps to be maintained, and the need for reverse engineering of these apps has been emphasized [8, 9].

For the effective maintenance of a web app, we need to understand how it utilizes its web components to create its web pages [1, 10, 11]. This means that an abstract model is required to represent web page creation. In addition, if one wishes to employ such a model for the efficient maintenance of a web app, it is necessary to extract the model from the web app automatically.

When a page request occurs in a web browser, multiple web components, such as JavaServer Pages (JSPs) and servlets, interact on the web server. During this process, web components access database resources such as tables and attributes for storing, updating, deleting, or retrieving persistent data. When the process is finished, the resulting page is created and returned to the browser. Therefore, the model that represents the page creation process needs to express the relationships between the web components and database resources for the effective understanding and maintenance of the web app.

Previous studies [1, 10-12] have proposed methods to define and extract models to represent such page creation processes. These studies have utilized dynamic [10] or static...
analyses to extract Java web components and their interactions. That is, Java web components and their interactions were extracted by only analyzing the source code (in a static analysis) or by executing web apps (in a dynamic analysis). However, these studies have not extracted interactions between database resources and web components.

If a model for the page generation process includes interactions between web components and database resources, unlike in previous studies, that model will yield the following advantages.

First, entities that depend on database resources can be identified at a more semantic level in terms of software architecture. Some previous studies [1, 10-12] have not identified database resources. Others (described in detail in Subsection II-B) have identified database resources but considered the base entity that interacts with database resources as a source file unit. For example, one model identifies the location (file name and line number) of the source code that accesses a database [13]. Our approach allows web developers to understand such dependencies on a per web component basis, so that web apps can be managed at a more abstract level.

Second, when changing requirements, or database resources in particular, the ripple effects can be analyzed in more abstract units, i.e., on a web component basis rather than a source file basis. Third, web engineers can analyze and manage access to database resources by web components and attempt to improve quality aspects, such as the performance, in terms of software architecture. Fourth, when performing an architecture conformance check [14] on a web app, we can check whether architecture inconsistencies [14, 15] occur in terms of web components and database resources, so that such tests can be more effective.

This paper proposes a page generation graph model that represents Java web components, database resources, and their interactions. We also propose a dynamic analysis method to automatically extract this model from a Java web app. We perform experiments with open source web apps to verify the validity of the proposed model and extraction method.

In order that our proposed method is less influenced by the introduction of new technologies (such as programming languages and development frameworks), the method dynamically extracts the page generation graph model. The key responsibility of our method is to identify when web components access database resources. To this end, our method dynamically observes the database-related objects provided in the Java standard library.

This paper describes the background, page generation graph model, and model extraction method, and analyzes the application of the extraction method, respectively. Finally, our conclusions and directions for future research are described.

II. BACKGROUND

Before introducing our technique, we describe existing models [1, 10-12] that represent interactions between web components. In particular, we describe the collaboration model introduced in [10] in further detail, which the graph model proposed here is based on and extends. Second, we examine previous studies dealing with interactions between apps and databases.

A. Collaboration Model

A user usually clicks a hyperlink on a web browser to request a new web page. When such a request occurs, various web components (such as JSPs, servlets, and Hypertext Markup Language (HTML) pages) interact with each other within the web server. Then, a new page is created and returned to the web browser. To represent these interactions, the research in [10] proposed the collaboration model, which is a model for handling a page request and is represented by a directed graph. An example of the collaboration model is presented in Fig. 1(a), which shows the interactions between web components when clicking a hyperlink to read book information from the Online Bookstore app (http://www.hanbit.co.kr/store/books/look.php?p_code=B5476653696). Fig. 1(b) shows the generated page. The Online Bookstore app is an open source Java web app that provides book information and the functionality to write and read book reviews.

The interactions shown in Fig. 1(a) can be described as follows. When a user makes a request to read book information, this request is delivered to a web container on the web server. The web container invokes the books-info servlet, which acts as a controller [16] to handle this request. The books-info servlet performs business logic related to retrieving the book information. Then, the servlet forwards the request to the WebTemplate.jsp component, which acts as a view [16] to generate the results page. This JSP includes LoginWindow.html in the upper left corner of the page in Fig. 1(b) to display the user interface to log into an account. In addition, this JSP includes BooksInfoView.jsp to display the book information in the form of a table in the center of the screen. The generated page is delivered to the web browser through the web container.

When a user requests a page, web components typically interact with each other on the web server. Web components access database resources for persistent data manipulation during this process. Therefore, a model that represents the page creation process needs to express the relationships between web components and database resources for the effective understanding and maintenance of web apps. However, in [10] database resources are not represented in the collaboration model, because the authors only considered interactions in terms of web components.
For effective maintenance, it is necessary to identify and document the database resources required to process a user's page request. Therefore, we propose the model depicted in Fig. 2. Fig. 2 shows why each web component requires a particular database resource on a per web component basis. This example illustrates interactions that take place on the web server when requesting book information, as shown in Fig. 1. The difference from Fig. 1 is that we can understand that the books-info servlet retrieves the goodsinfo table to obtain book information, and subsequently forwards this information to the WebTemplate.jsp component for page creation.

Other previous studies [1, 11, 12] have proposed models for representing the interactions between web components on a web server, similar to [10]. The studies in [1, 10] extract their models through dynamic analyses, and those in [11, 12] extract their models through static analyses. However, like [10], the approaches [1, 11, 12] did not consider database resources or database access.

This present study aims to generate the graph model shown in Fig. 2 through a dynamic analysis. In other words, it represents the dependency relations between web components and database resources.

For effective maintenance, it is necessary to identify and document the database resources required to process a user's page request. Therefore, we propose the model depicted in Fig. 2. Fig. 2 shows why each web component requires a particular database resource on a per web component basis. This example illustrates interactions that take place on the web server when requesting book information, as shown in Fig. 1. The difference from Fig. 1 is that we can understand that the books-info servlet retrieves the goodsinfo table to obtain book information, and subsequently forwards this information to the WebTemplate.jsp component for page creation.

Other previous studies [1, 11, 12] have proposed models for representing the interactions between web components on a web server, similar to [10]. The studies in [1, 10] extract their models through dynamic analyses, and those in [11, 12] extract their models through static analyses. However, like [10], the approaches [1, 11, 12] did not consider database resources or database access.

This present study aims to generate the graph model shown in Fig. 2 through a dynamic analysis. In other words, it represents the dependency relations between web components and database resources.

B. Identification of Database Access

According to [17], the purposes of identifying access to a database in an app include detecting Structured Query Language (SQL) errors, enhancing database-related security, and aiding database evolution. Below, we briefly review existing studies for these purposes.

First, some studies have identified SQL statements that access a database. Following identification, they analyze whether these statements are syntactically or semantically correct. These include [18] and [19]. Second, some studies [20, 21] have identified and analyzed query statements that access a database, determining if these statements are vulnerable to SQL injection attacks, and so on.

Third, the study in [22] proposed a method to identify statements that access a database from a Java web app. Unlike [22], we aim to identify web components that access a database, rather than statements. The difference can be explained as follows. Suppose that a statement \( s \) in a method \( m \) accesses a database resource \( r \). The goal in [22] is to statically identify \( s \). However, \( m \) can be called dynamically by a web component \( c_1 \) or \( c_2 \). In our study, if \( c_2 \) calls \( m \), then we can identify a dependency relation between the web component \( c_2 \) and database resource \( r \).

The study in [23] proposed a method to identify statements that access a database from a Java web app. The authors also proposed a metric for measuring the complexity of generating SQL statements in the app. Like [22], they identify statements but not web components.

The study in [24] proposed a method of extracting the constraints of a database schema by identifying and analyzing statements that access a database from the source code. This research can help to ensure that the co-evolution of the source code and the database schema of an app occurs consistently.

These previous studies have not addressed the problem of identifying web components that access database resources. We will discuss this problem for a Java web app.
III. DYNAMIC EXTRACTION OF A PAGE GENERATION GRAPH INCLUDING DATABASE USAGE

This study aims to generate a graph model like that depicted in Fig. 2 through a dynamic analysis. In other words, it represents not only interactions between web components on a web server, but also dependency relations between web components and database resources.

A. Page Generation Graph Including Database Usage

We propose a graph model page generation graph including database usage (PG2ID) in Definition 1 to represent the web components, database resources, and dependency relations between them required to create a page on a web server. This graph model extends the collaboration model proposed in [10]. Fig. 2 presents an instance of this graph model.

(Definition 1) PG2ID: Directed graph $G_r(V_r, E_r)$ for request $r$. Given a page request $r$, the page generation graph including database usage, $G_r(V_r, E_r, t, o)$ of $r$, is a direct graph satisfying the following:

- $V_r$ is a set of web components (such as JSPs, servlets, and HTML pages) and database resources (such as tables and attributes).
- $E_r$ represents the relations between nodes included in $V_r$. For $v_i$ and $v_j$ belonging to $V_r$, when $v_i$ invokes $v_j$, $<v_i, v_j>$ is included in $E_r$. There are 10 types of calls or relations (i.e., Enter, Forward, Include, Redirect, Select, Insert, Update, Delete, Create, and Drop), which are described in detail below.
- $t: E_r \rightarrow \{\text{Enter, Forward, Include, Redirect, Select, Insert, Update, Delete, Create, and Drop}\}$, $t$ is a function to determine the type of each relation contained in $E_r$.
- Descriptions of the calls are as follows.
  - Select: Web component $v_i$ uses database resource $v_j$ for information retrieval (i.e., $\text{select}$ query statements).
  - Insert: Web component $v_i$ uses database resource $v_j$ for storing information (i.e., $\text{insert}$ query statements).
  - Update: Web component $v_i$ uses database resource $v_j$ to update information (i.e., $\text{update}$ query statements).
  - Delete: Web component $v_i$ uses database resource $v_j$ to delete information (i.e., $\text{delete}$ query statements).
  - Create: Web component $v_i$ creates a new table $v_j$ (i.e., $\text{create table}$ statements).
  - Drop: Web component $v_i$ drops table $v_j$ (i.e., $\text{drop table}$ statements).
- $o: E_r \rightarrow \{1, 2, \ldots, |E_r|\}$. $o$ is a function that returns a calling sequence for each relation included in $E_r$. This function represents the order in which web components and database resources are executed.

B. Extraction of a Page Generation Graph Including Database Usage

1) Overview of Dynamic Extraction Method

This study proposes a novel dynamic analysis method for extracting PG2IDs from Java web apps. We extend the method proposed in [10], which extracts interactions between Java web components occurring on a web server during page creation and expresses the interactions through the collaboration model.

The main aim of this extension is to extract and add database resources to the collaboration model as nodes, and to identify and add Select, Insert, Update, Delete, Create, and Drop relations between web components and database resources as edges.

Relations between web components are identified using the Java servlet filter and wrapper standard application programming interface (API) [25] according to the method proposed in [10]. However, new requirements (i.e., to understand which web components access which database resources) cannot be satisfied using just the method proposed in [10]. Therefore, this study applies the proxy design pattern [26], and slightly instruments the source code of a web app to identify database resources and $\text{Select}$, $\text{Insert}$, $\text{Update}$, $\text{Delete}$, $\text{Create}$ and $\text{Drop}$ edges.

The key idea for identifying when a database is accessed is to dynamically observe database-related objects provided in the Java standard library. To achieve this, we decided to employ the proxy pattern, which can be applied to control access to a desired object and add functionality to an object when the object is executed.

2) Analysis of JDBC API

The Java Database Connectivity (JDBC) API is provided by the Java standard library for database access. In general, the procedure for a web app to access a database using the
JDBC API progresses as follows. A web app attempts to connect to a database in step 1 of Fig. 3(a). If the connection to the database is successful, then the Connection object is returned. To read data from the database, the Connection object must first be called on the Connection object, and then the Statement object is obtained (step 2). The data in the database can then be retrieved by calling the executeQuery() method on the Statement object (step 3). Alternatively, in step 3, the data in the database can be updated by calling the executeUpdate() method on the Statement object.

This study detects the execution of executeXXX() type methods, such as executeQuery() and executeUpdate(), to identify which web component accesses which database resource. Then the argument of the executeXXX() method is parsed to identify target database resources (such as tables and attributes) and determine the type of relation (such as Select, Delete, Update, or Insert). In addition, we utilize the standard filter and wrapper API of a Java servlet to identify its source web component for a relation, i.e., the web component that requests the execution of the executeXXX() method above.

3) Dynamic View of Extraction System

Fig. 4 presents a sequence diagram representing the operation of our system for extracting PG2IDs from web apps. We place a proxy object at each step in Fig. 3(a) above and ultimately detect the execution of executeXXX() methods. The main elements of the extraction system are described in the

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/webdb", "user", "1111"); // step 1
Statement stmt = conn.createStatement(); // step 2
ResultSet rs = stmt.executeQuery("select * from goodsinfo where code = " + code + ";"); // step 3
(a) Java code using the JDBC API to execute a SQL query

Connection conn = DriverManagerProxy.getConnection("jdbc:mysql://localhost:3306/webdb", "user", "1111"); // step 1
Statement stmt = conn.createStatement(); // step 2
ResultSet rs = stmt.executeQuery("select * from goodsinfo where code = " + code + ";"); // step 3
(b) Modified Java code to understand dependency relations with database resources
```

Fig. 3. Original code that accesses a database using the JDBC API and the modified code to identify dependencies with the database.

Fig. 4. Dynamic view for extracting dependency relations between web components and database resources.
order utilized to process a database request.

In step 1 of Fig. 4, a web app attempts to connect to a database using the DriverManagerProxy class corresponding to the DriverManager class. The proxy class then creates a Connection object with the help of its original DriverManager class. The proxy then creates and returns a ConnectionProxy object, which corresponds to the Connection object.

The ConnectionProxy class implements the Connection interface, because the proxy must act as the Connection interface. At this point, delegation is applied to increase the reuse of existing source code. In other words, to define a method of the Connection interface, ConnectionProxy handles a request by calling the corresponding method of the original Connection object. An example is presented below.

In step 2, when the web app calls the CreateStatement() method of the ConnectionProxy object to access data in the database, its original Connection object is utilized to create a Statement object, and the corresponding StatementProxy object is created and returned.

In step 3, when the web app requests data access to the StatementProxy object using an executeXXX() method, we detect this database access. After detection, the StatementProxy object uses the original Statement object to work with the database. Then, our extraction system performs database usage analysis.

During this analysis, the StatementProxy object first identifies the web component currently calling the executeXXX() method. For this identification, we extend the method of extracting the collaboration model proposed in [10].

In [10], when a web component is executed, a logger that records interactions between web components records the start and end of the execution, as well as the call type (Enter, Forward, Include, or Redirect) of the web component. Our novel idea for extending this is that when accessing a database, the logger also records the database access (e.g., accessed database resources and utilized SQL statements). This allows the identification of web components that access databases.

For example, suppose that a web component c1 is executed as the first web component by a user request, and a web component c2 is included in c1 while c1 is executing. According to the notations in [10], the collaboration model for this request can be expressed as: (Enter c1 (Include c2)). If c2 calls executeQuery("select * from person;") during its execution, then the StatementProxy object logs this database access information before c2 completes its execution. This collaboration model appears as follows: (Enter c1 (Include c2 (Select person))). On the other hand, if c1 calls the same method after including c2, then the StatementProxy object also detects this database access, and this collaboration model appears as follows: (Enter c1 (Include c2) (Select person)).

It can be observed in the above example that the web component performing the database operation is that of the innermost block of the block, which represents the operation in the collaboration model. It is important to note that the collaboration model itself does not contain database-related notations (e.g., for SELECT query statements, tables, and attributes). However, for the ease of explanation in the previous paragraph we extended the notations of this model to include database-related information.

From a software design perspective, the method proposed in [10] to extract the collaboration model can be described as follows. Logging operations are continuously performed while a page request is processed. Therefore, the request wrapper [25] is employed as a storage area for logging, because the scope of the request wrapper is the request processing period. A servlet filter [25] is utilized to determine when a web component starts and ends execution.

In this study, we propose a novel method to identify web components that access databases, which detects execution of the executeXXX() method by applying the proxy pattern and records the detected execution in the above logging area. We also suggest using the StatementProxy object to understand which web components access which database resources and why these accesses occur. The StatementProxy object parses the argument of the detected executeXXX() method to identify the database resource and relationship type. This parsing can be supported using open source tools such as the General SQL Parser (GSP) [27].

In terms of the time complexity, the most complex part of the approach proposed in this subsection is parsing a SQL statement that is passed as an argument of the executeXXX() method. In general, it takes O(n^2) to create a parse tree with a SQL statement, where n is the length of the statement [28, 29]. Thus, the time complexity of the proposed approach can be regarded as O(n^3).

4) Static view of extraction system

Fig. 5 illustrates the classes and interfaces that make up our extraction system and the relationships between them in a class diagram.

Because each proxy class p must employ the functionality of its original class or interface s, p has an association or a dependency relation with s. For example, the ConnectionProxy class has an association relation with the Connection interface.

In the process of utilizing the JDBC API, an object of the previous step creates an object of the next step. Similarly, in our study the proxy of step i generates the proxy of step i + 1. In addition, the proxy of step i + 1 is created using its corresponding original object. Thus, the proxy of step i has dependency relations with the original class of step i + 1 and the proxy class of step i + 1. For example, the DriverManagerProxy class has dependency relations with the Connection interface and ConnectionProxy class.
Web engineers aim to minimize modifications when modifying web apps for database usage analysis. To make this possible, the proxy classes `ConnectionProxy` and `StatementProxy`, which are used in the form of objects, implement the original interfaces. Thus, as can be observed by comparing steps 2 and 3 in Figs. 3(b) and 3(a), web engineers do not need to modify the code of web apps, but the proxies are used internally in these steps for database usage analysis.

Web apps may call an `executeXXX()` method using subinterfaces of `Statement` rather than `Statement`. In this case, two additional proxies are provided to identify the relationships between database resources and web components. That is, `Statement` has two subinterfaces: `CallableStatement` and `PreparedStatement`. For these two interfaces, we implement `CallableStatementProxy` and `PreparedStatementProxy` classes, like `StatementProxy`, which is implemented for `Statement` using delegation.

Web engineers perform the following installation and modification tasks to extract PG2IDs from web apps. They must first install the module developed in this study (see Fig. 5). Thereafter, in the source code of their web apps they only need to modify step 1 of Fig. 3(a) to step 1 of Fig. 3(b).

**IV. EXPERIMENT AND ANALYSIS OF RESULTS**

This study tests three open source web apps (Online Bookstore, JMBoard, and Online Movie Ticket Booking) to verify the feasibility of our extraction method. The reasons (similar to those described in [1]) for choosing these web apps are as follows. First, these were used to verify the feasibility and validity of previous studies [1, 10, 30-34] in web engineering. Second, the source code is available from popular open source repositories. Third, these web apps are composed of HTML, JavaScript, servlets, JSP, standard action tags, and Asynchronous JavaScript and Extensible Markup Language (XML) (AJAX).

The Online Bookstore and JMBoard apps display web pages in Korean. To assist the reader's understanding, we translate the Korean web pages into English to show the experimental results.

Fig. 6(a) shows the interactions between web components and database resources when a user requests a list of posts from the Online Bookstore app. Fig. 6(b) shows the results page generated for this request.

The interactions shown in Fig. 6(a) can be described as follows. When a user makes a request to read a list of posts, the request is delivered to the server's web container. This web container calls the `bbs-list` web component, and this component performs business logic related to the retrieval of a list of posts. `bbs-list` then forwards the current request to the `WebTemplate.jsp` web component, and this JSP generates the results page.

`bbs-list` accesses a database once while performing the business logic. This component accesses the `bbs` table to obtain information on the list of posts. By comparing Fig. 2 with Fig. 6(a), we can observe that the Online Bookstore app shares the `WebTemplate.jsp` component to display the web pages for the two requests but uses a different table for each request.
of them.

Second, the JMBoard app (http://happycgi.com/8168) is a web app that allows the creation and deletion of bulletin boards, and the posting of messages on generated bulletin boards. Fig. 7(a) shows the interactions between web components and database resources when a user makes a request to create a bulletin board named mTest, and Fig. 7(b) shows the results page generated for this request.

When a user requests to create a bulletin board, the request is delivered to the server's web container. The web container calls the web component admin.jsp. Thereafter, this component performs business logic related to the creation and generates the results page.

When a user requests to create a bulletin board, the request is delivered to the server's web container. The web container calls the web component admin.jsp. Thereafter, this component performs business logic related to the creation and generates the results page.

The admin.jsp component accesses a database four times while performing the business logic. First, this component creates the mTest and mTest_cmt tables to store messages posted to the created bulletin board. Second, the board_env table is accessed to set the environment information, such as the name and background color of the board. Third, information is retrieved from the board_env table to list the currently generated bulletin boards. From Fig. 7(b), we can observe that the currently generated bulletin boards are mTest, research, university, and web. This identification of interactions can help to reengineer web apps (for example, to cope with changes in database schemas), as mentioned in [2, 3, 5, 6, 11].

Fig. 8 shows the result of asking the JMBoard app to read a post on the mTest bulletin board. To generate the resulting page in Fig. 8(b), web components and database resources interact as shown in Fig. 8(a). Upon receiving the user's request, the JMBoard controller performs business logic to obtain information about the post. The JMBoard component then forwards the request to the jmboard.jsp view to produce the resulting page. This JSP component includes several web components (i.e., top.jsp, view.jsp, and bottom.jsp) for generating the results page.

It can be observed in Fig. 8(a) that the JMBoard component accesses a database more than 10 times while performing the business logic. The board_env table is accessed four times, and the mTest_cmt table is accessed once. The mTest table is accessed in other cases. In the cases of board_env and mTest_cmt, all their attributes are retrieved using SELECT query statements. In the case of mTest, either all or only certain of its attributes (i.e., filename, filetype, filesize, and did) are retrieved. In other cases, the number of records matching some condition is counted. It should be noted here that the JMBoard component issues the same query multiple times. For example, this component executes the query 

SELECT * FROM mTest WHERE no = '1' three times during the interactions shown in Fig. 8(a). This may lead to performance issues, because only SELECT query statements are executed while processing this request. This demonstrates that our proposed dynamic analysis can help in the identification of performance improvements.

It should also be noted that for the scenarios in Fig. 7 and Fig. 8 the JMBoard app shares the mTest, mTest_cmt, and board_env tables. This example shows that our method can identify web components that share database resources (in this case, the admin.jsp and JMBoard components).

Third, the Online Movie Ticket Booking app (https://sourceforge.net/projects/onlinemovietick/?source=directory) is a system that allows one to browse movie information and book movies. When a user selects a movie on the page in Fig. 9(a), specific information on that movie can be observed, as shown in Fig. 9(b). In this case, the user selects the WALL-E movie.

To generate the resulting page in Fig. 9(b), web components and database resources interact as shown in Fig. 9(c). Upon receiving the user's request, the filmdesc.jsp web component performs business logic to obtain the movie information. Thereafter, this component itself creates the results page. The filmdesc.jsp component accesses a database once during execution of the business logic. This component retrieves all the attributes for the WALL-E movie from the film table.

PG2IDs enable database usage analysis at the attribute level of a database table. In other words, the dependencies
Automatic Extraction of Dependencies between Web Components and Database Resources in Java Web Applications

between web components and attributes can be represented in PG2IDs. The example shown in Fig. 10 demonstrates the possibility of such an analysis.

As another example from the Online Movie Ticket Booking app, the login results are shown in Fig. 10(b). If a user enters an ID and password, attempts to log in, and succeeds, the results page shown in Fig. 10(b) is displayed. In this case, the user logs in using the Admin ID.

To generate the results page, web components and database resources interact as shown in Fig. 10(a). Upon receiving the user's request, the login.jsp component executes business logic for user authentication. Subsequently, this component itself creates the results page.

The login.jsp component accesses a database three times while performing the business logic, as shown in Fig. 10(a). First, this component obtains attributes (such as passw and id) from the Customer table to verify that the entered ID and password are valid. Second, it increments the value of the attribute noofvisits, which indicates the number of visits in the Customer table. Third, the component obtains additional attributes (such as the user name) from the Customer table.

By comparing Fig. 9 with Fig. 10, we can observe that the Online Movie Ticket Booking app does not share web components or database resources for the two requests. Therefore, owing to the low degree of coupling between the modules handling the two requests an independent design and implementation is possible, which makes it relatively convenient to maintain the modules.

The case study employing these three open source web apps reveals the following. First, dependency relations with database resources can be determined at the web component level. Therefore, it is possible to understand web apps at a more abstract level than the source code level (such as at the method, class, or file level). This implies that one can understand web apps from a software architecture perspective. Second, it is possible to identify web components and web pages that are affected by database resource or schema changes. Third, it is possible to identify web components that
share database resources. As described above, if our method is employed then the understanding of web apps is improved, and so maintenance of web apps can become easier.

V. CONCLUSION

This paper proposes a graph model that represents the web components, database resources, and relations between them required to generate a page from a web server. We also propose a dynamic analysis method for automatically extracting this model from a Java web app. This method applies the proxy pattern and Java servlet filter and wrapper API to understand which web components access which database resources.

The contributions of this paper are summarized as follows. First, unlike previous research, we consider base entities that interact with database resources as web components (e.g., servlets and JSPs). Second, we propose a method for automatically extracting these interactions from Java web apps. Third, because the proposed reverse engineering method utilizes a dynamic analysis, it is less affected by the introduction of new technologies. Fourth, a case study on open source web apps demonstrated that the proposed method can be applied to the understanding and maintenance of web apps.

In future research, we will explore how to effectively visualize page generation graphs including database usage. We will also consider application frameworks such as the Spring framework.

ACKNOWLEDGEMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2017R1D1A1B03029374). This work was also supported by the Catholic University of Korea, Research Fund, 2018. The present research has been conducted by the Research Grant of Kwangwoon University in 2019.
REFERENCES


Jaewon Oh
received his BSc, MSc, and PhD in Computer Science from Seoul National University, Korea in 1997, 1999, and 2004, respectively. He was a senior research engineer at Samsung Electronics Company from 2004 to 2007. Currently, he is a full professor at the School of Computer Science and Information Engineering of the Catholic University of Korea. His current interests include web engineering, software evolution, and software engineering.

Woo Hyun Ahn
received a B.S. degree in the Division of Electrical Engineering from Kyungbook National University in 1996, and M.S. and Ph.D. degrees from the Division of Electrical Engineering at the Korea Advanced Institute of Science and Technology (KAIST) in 1998 and 2003, respectively. He worked in the software research center of Samsung Electronics Co. Ltd from 2003 to 2005. He is currently a full professor in School of Software at Kwangwoon University. His research interests include operating systems, embedded systems, and web engineering.

Taegong KIM
received his BS degree in Computer Science and Statistics from Seoul National University, Korea in 1983. He also received his MS and PhD degrees in Computer Science and Statistics from Seoul National University in 1985 and 1994, respectively. He is currently an associate professor in the Department of Computer Engineering, Inje University, Korea. His main interests include software engineering, refactoring, and code smell detection.