2017 IEEE 31st International Conference on Advanced Information Networking and Applications

A Proposal of Software Architecture for Java Programming Learning Assistant System

Nobuya Ishihara, Nobuo Funabiki, Minoru Kuribayashi
Department of Electrical and Communication Engineering
Okayama University, Okayama, Japan
Email: {n.ishihara,funabiki,kminoru}@okayama-u.ac.jp

Wen-Chung Kao
Department of Electrical Engineering
National Taiwan Normal University, Taipei, Taiwan
Email: jungkao@ntnu.edu.tw

***Abstract*—To improve Java programming educations, we have developed a Web-based *Java Programming Learning System (JPLAS)*. To deal with students at different levels, JPLAS provides three levels of problems, namely, *element fill-in-blank problems*, *statement fill-in-blank problems*, and *code writing problems*. Unfortunately, since JPLAS has been implemented by various students who studied in our group at different years, the code has become complex and redundant, which makes further extensions of JPLAS extremely hard. In this paper, we propose the software architecture for JPLAS to avoid redundancy to the utmost at implementations of new functions that will be continued with this JPLAS project. Following the MVC model, our proposal basically uses Java for the model (M), JavaScript/CSS for the view (V), and JSP for the controller (C). For the evaluation, we implement JPLAS by this architecture and compare the number of code files with the previous implementation.**

I. Introduction

Java has been extensively used in industries and educated in schools. To improve Java programming educations, we have developed a Web-based Java Programming Learning Assistant System (JPLAS) that can assist self-studies of students while reducing workloads of teachers [1]- [4]. JPLAS has three types of problems to accommodate a variety of students at different learning levels. The first one is the *element fill-in-blank problem* which requires students to fill in correct elements in the blanks in the given Java code. Next, the second one is the *statement fill-in-blank problem* which makes students write whole statements that are blank in the code. In the third one, *code writing problem* the task of students is to write whole Java codes to satisfy the given specifications described by the test code. The difficulty level is designed to increase in this order of the three problems.

As a laboratory project, JPLAS has been continuously implemented and modified by plural students at different years in our group. Furthermore, several functions in JPLAS have been extended nearly every year. Through this project, we expect that students have experienced programming for practical systems that have been used in Java programming educations in universities. Unfortunately, JPLAS has many redundant classes and methods in the code that have substantially the same functions. Because most of the students did not have sufficient knowledge and experiences on Java programming and JPLAS implementations, they implemented new functions by copying the whole existing code and modifying/adding the related part. They commented out or did not call unnecessary parts of the code that remain in the JPLAS code. As a result, a large number of *clone codes* [5] have been accumulated in the code of JPLAS which make it long and redundant.

In this paper, we propose a *software architecture for JPLAS* that can avoid clone codes to the utmost even when new students implement new functions in JPLAS. This architecture strictly follows the *MVC model* that is the common architecture for Web application systems including JPLAS. Our software architecture basically uses *Java* for the *model (M)*, *JavaScript/CSS* for the *view (V)*, and *JSP* for the *controller (C)*. It is emphasized that *Servlet* is not used in our proposal to avoid the possible redundancy that can happen between Java codes and Servlet codes where the same function can be implemented. More specifically, in the model, a design pattern called *responsibility chain* is adopted to handle the three problems in JPLAS, and the functions for database access are implemented, where the controller does not handle. In the view, the user interface is dynamically controlled with *Ajax*, which can reduce the number of JSP files.

For the evaluation of our proposal, we implement JPLAS from scratch by following this software architecture. First, we compare the number of code files with the previous implementation, which shows that the number of code files becomes 25%. Then, we count the number of newly added code files when two functions are added to JPLAS, which shows the number of new files is small for either new function.

The rest of this paper is organized as follows: Section II reviews the outline of the current JPLAS and notes its problems. Sections III and IV present the software architecture for JPLAS and the implementation, respectively. The evaluations are shown in Section V show evaluations. Section VII concludes this paper with future works.

II. Review of Current JPLAS

In this section, we review the outline of current *JPLAS*.

A. Software Platform

In JPLAS, *Ubuntu* is adopted for the OS of JPLAS running on *VMware*. *Tomcat* is used as a Web server using *JSP/Servlet*. *JSP* is a script with embedded Java code within the HTML code, where Tomcat can return a dynamically generated Web page by JSP to the client. *Servlet* is a Java code that can

1550-445X/17 $31.00 © 2017 IEEE
DOI 10.1109/AINA.2017.56

dynamically generate a Web page. *MySQL* is adopted as a database for managing the data.

B. Three Problems in JPLAS

In JPLAS, the three types of problems are provided. For each problem, JPLAS has functions which not only teachers to generate and register new problems but assist students in answering them.

1) Code Writing Problem: This problem asks students to write a whole code from scratch that satisfies the specifications given by a *test code*. The correctness of the code written by a student is marked by using this test code on *JUnit* that is an open source software for the test-driven development (TDD) method [6]. A teacher needs to prepare the specification and the test code to register a new assignment in JPLAS.

2) Element Fill-in-blank Problem: This problem requires students to fill in the blank elements in a given Java code. The correctness of the answers are marked by comparing them with their original elements in the code. Thus, the original elements must be the unique correct answers for the blanks. To help a teacher to generate an element fill-in-blank problem, we have proposed a *blank element selection algorithm* [2] [3].

3) Statement Fill-in-blank Problem: This problem asks students to fill in the blank statements in a given Java code. The correctness of the answers are marked by using the test code on *JUnit* as the code writing problem, where the combined code with the blank one and the answers is tested. To help a teacher design a statement fill-in-blank problem, we have proposed a *blank statement selection algorithm using the program dependency graph (PDG)* [4].

C. Support Functions for Teacher

JPLAS provides several support functions for teachers to generate and register new problems.

1) Code Writing Problem: JPLAS allows a teacher to register a new code writing problem with the test code, and register a new assignment by selecting problems and filling in the statement.

2) Element Fill-in-blank Problem: JPLAS allows a teacher to register a new Java code, generate a new element fill-in-blank problem using the algorithm, and register a new assignment by selecting problems and filling in the statement.

3) Statement Fill-in-blank Problem: JPLAS allows a teacher to register a new Java code, generate a new statement fill-in-blank problem using the algorithm, register the test code, and register a new assignment by selecting problems and filling in the statement.

4) Markers in Database: In the database, each problem has been stored in a single text field as a string with the markers in Table I:

D. Support Functions for Student

Likewise, JPLAS provides several support functions for students to answer problems. The answer from a student is processed in JPLAS by the following steps:

TABLE I
MARKERS IN PROBLEM TEXT.

Marker	description
//@JPLAS answer	the following paragraph contains correct answers
//@JPLAS statement	the following paragraph contains the problem statement
//@JPLAS test code	the following paragraph contains the test code
//@JPLAS output	the following paragraph contains the output of the code
__ (under line)	the corresponding element is blanked
//@JPLAS blank	the following statement is blanked
null	problem code

(1) When a student accesses to JPLAS, the list of the assigned problems to the student is displayed.
(2) When a problem is selected by the student, the corresponding problem text in the database is displayed using the marker in Table I.
(3) The student fills the answers in the corresponding forms.
(4) The answers submitted by the student, they are marked in the server, and both the answer and the marking results will be saved in the database.
(5) JPLAS offers feeds back to the student.
(6) If necessary, the student can repeat the steps from (3).

Figure 1 illustrates the overview of the problem answer procedure.

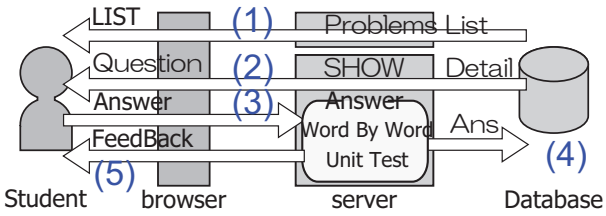

Fig. 1. Overview of problem answer procedure in JPLAS.

E. Marking Functions

Each problem possesses a unique marking function in the server.

1) Code Writing Problem: In the code writing problem, the following two-step making is applied to the code answered by a student after creating a working folder in the server. In the first step marking, the syntax of the answer code is verified by compiling it. The student will be informed of the syntax errors if they are found. When the compile succeeds, the score of one hundred is given to the student, and the second step is applied. Otherwise, the point of zero is given. In the second step marking, the logic or behavior of the code is tested by the unit test on *JUnit*. The error message from *JUnit* will be returned to the student if any test items in the test code fail. The marking point is given by the rate of the cleared test items among all the items.

2) Element Fill-in-blank problem: In the element fill-in-blank problem, each answer from a student is compared with the corresponding correct word stored in the database. Students

will know the correctness of their answers by the change of the background color of each blank. The marking point is given by the rate of the cleared blanks among them.

3) Statement Fill-in-blank Problem: In the statement fill-in-blank problem, first, the answer from a student is inserted into the blanks in the problem code, to make a complete answer code. Then, the same two-step making as in the code writing problem is applied to this answer code. The marking point is also given by the rate of the cleared test items. The correctness of the statements answered by the student is as well reflected in the change of the background color of each blank.

F. Drawbacks in Implementation

As discussed in Section I, JPLAS has been implemented by plural students who studied in our group at different years. In addition, each of the three problems has been implemented by a different student. The code for the code writing problem was first implemented by a student who started the JPLAS project. Then, the code for the element fill-in-blank problem was implemented by another student through copying this existing code and modifying it. The code for the statement fill- in-blank problem was implemented by another student in the similar way.

Therefore, a plenty of *clone codes* have been accumulated in the codes for JPLAS and made them long and redundant. Besides, the documents for the code implementations are not properly managed. Furthermore, these implementations did not follow the *MVC model* strictly and did not consider new Web technologies such as *Ajax*, *HTML5*, and *CSS3*, where Java is used for Model, JSP is for View, and Servlet is for Controller as shown in Figure 2. Some functions in Model controls View directly without Controller. As a result, it is challenging for new students to make further extensions due to the limited time and skills.

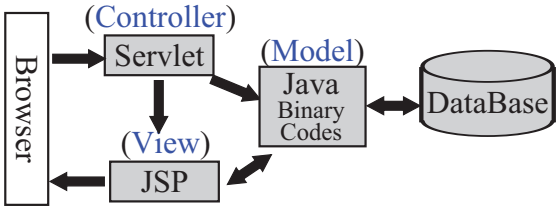

Fig. 2. Languages for MVC model in existing JPLAS.

In JPLAS, a lot of user interfaces have common items such as the menu. Unfortunately, in the existing codes for JPLAS, the corresponding code part to the common ones is always included in the JSP code for each interface, which causes the redundancy of codes. Besides, students need sufficient time to learn the programming by Servlet and the setup of Tomcat for the use. Thus, it is better to implement JPLAS without Servlet.

III. Proposal of Software Architecture for JPLAS

In this section, we present the software architecture for JPLAS to solve the drawbacks in the current JPLAS implementation.

A. Adopted Languages for MVC Model

This software architecture strictly follows the MVC model, because it is a standard architecture for a Web application system. As illustrated in Figure 3, we use Java for the model, HTML/JavaScript (JS) for the view, and JSP for the controller. By using different languages for each of the MVC model, it is expected to effectively avoid redundant descriptions of the same function called *code clones*.

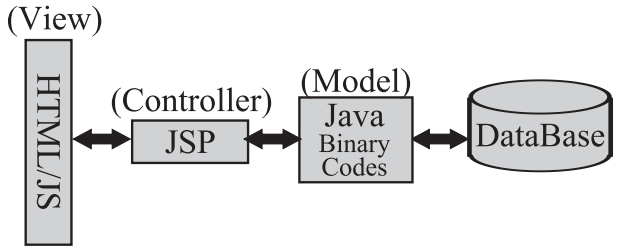

Fig. 3. Languages for MVC model in new JPLAS.

By adopting HTML, CSS, and JavaScript for View, advanced user interfaces using animations and dynamic content changes can be implemented in JPLAS. By adopting JSP for Controller, loads of students for learning Servlet and Tomcat configurations can be reduced.

B. Software Architecture for Model

In the proposed software architecture, the model implements the logic functions of JPLAS by Java. For the independence from the view and the controller, any input/output to/from the model principally uses a string or its array that does not contain HTML tags.

1) Database Access: In the MySQL database server, plural databases can be defined, where each database can have plural tables. JPLAS uses one database to store the information on users, problems, and answers in the corresponding tables. To avoid the redundant codes by implementing common functions in database access into the upper classes, we prepare three classes to handle it. Figure 4 shows the relationships between them.

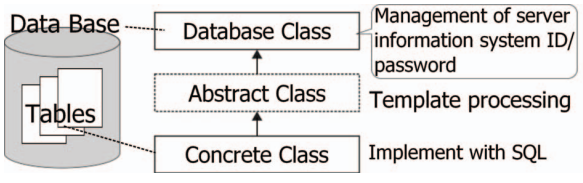

Fig. 4. Three classes for database access.

Database class manages the information necessary for database access, such as the server IP address, the port number, the database name, the system ID that is used to connect to the database server in JPLAS, and the password. The system

ID and the password are hidden in private variables in this class.

Table class makes a model of each table for users, problems, and answers so that it can be handled by Java. This table class handles the data access to/from the database using SQL commands. If the corresponding table does not exist, this class automatically generates the table using the initialization method.

Abstract class provides the common procedures for database access such as closing the data linkage, and unifying the data type for output. The table class implements this class.

2) Overview of Marking: JPLAS provides three different problems where each problem needs a different marking function. To automatically select the proper marking function to the answer from a student, the *responsibility chain* design pattern is adopted.

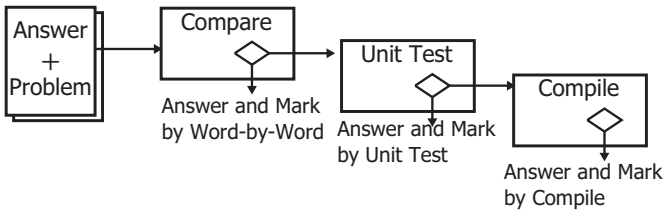

Fig. 5. Marking functions by responsibility chain.

As shown in Figure 5, by using the responsibility chain, the specified next marking process is automatically selected when the current process cannot handle the received data of the answer. The class for each marking process has the method to check whether the answer can be handled or not, the method to mark the answer if it can be handled, and the method to specify the next marking process if not. To give the integrated names to these methods, an abstract class is prepared such that any class for marking becomes its child class. Then, the methods in these marking classes have the names in the abstract class. For each marking class, the answer data from a student and the correct answer/test code from the database are the inputs, and the strings and the score are the outputs. The output strings are the data coming from the marking class to give students information to complete the answers. They include the compiler error messages, the JUnit output messages, and the marking results. The *word matching test* is specified as the first marking process.

3) Word Matching Test: When the data from the database contains the correct answer words of the problem, the *word matching test* is applied. In this marking, each answer is compared with the corresponding correct word in the database. When they are the same, it returns *correct*. Otherwise, it returns *wrong*. In this marking function, the grade is given by the rate of the number of correct answer words to the total number of correct answer words. Then, the *unit test* is specified as the next marking process.

4) Unit Test: When the data from the database contains the test code for the problem, the *unit test* is applied. In this marking, each answer code is first saved on the disk of the server with the test code. Then, the required commands to the unit test are called.

In the unit test, the answer code is compiled by the Java compiler. If it is successfully compiled, the test code is also complied. After that, the unit test using *Junit* is called by calling the command, and the standard output and the log file are obtained by strings. In this marking function, the grade is given by the rate of the number of successful tests to the total number of tests in the test code. Then, the *compiling test* is specified as the next marking process.

5) Compiling Test: When none of the previous tests were applied, the *compiling test* is applied. In this marking, each answer code is first saved on the disk of the server with the test code. Then, the command to compile the code is called. If it is successfully compiled, the score of one hundres will be given. Otherwise, the score will be zero. There is no next marking process.

C. Software Architecture for View

The *view* implements the user interfaces of JPLAS by using a CSS framework to provide integrated interfaces using *cascading style sheet (CSS)* in the Web standard. In this paper, *SkyBlue* [8]is adopted in the CSS framework, because it can provide proper layouts for multi-size displays without JavaScript codes.

1) Overview: Figure 6 illustrates the overview of the software architecture for the view. In a Web page for the user interface in JPLAS, the layout is described by HTML and CSS, where it consists of the title, the menu, and the main body of the assignment. The communication with the output of the server data to be displayed in the interface and the input data from a user to be sent to the server are handled by JavaScript for Ajax. As the feature of this architecture, the fixed parts of the interface are made by HTML with CSS, while the changeable parts of the interface are realized by JavaScript. By using different languages for the fixed parts and the changeable parts in the interface, this design can reduce the code size and simplify the code architecture.

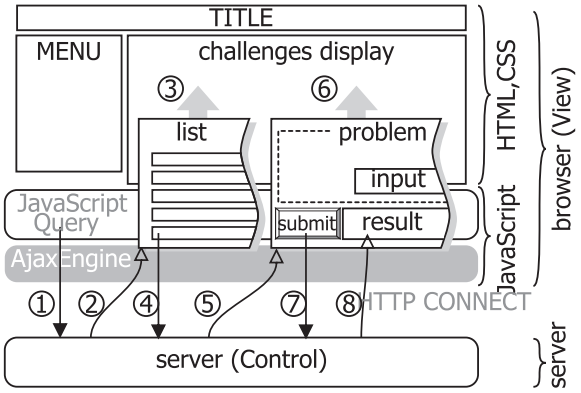

Fig. 6. Software architecture for view.

2) Communications for Answer Interface: The communications between the server and the browser for the assignment answer interface are as follows:

1) JavaScript in the browser requests the assignment list to JSP for the control in the server.
2) JSP on the server transmits the list in the table to JavaScript.
3) JavaScript updates the assignment list in the interface.
4) JavaScript requests the details of the clicked assignment submitted by the student to JSP.
5) JSP transmits the requested data after embedding the scripts for “input field”, ”answer button,” and ”result display field”.
6) JavaScript updates the interface.
7) When the student clicks the “answer button”, JavaScript collects the answers from the student, send them to the server, and requests marking them to JSP.
8) JSP on the server marks the answers.
9) JavaScript receives the marking results and shows them in the interface.

D. Control

The *control* in JPLAS is implemented by JSP. When it receives a request from the view, it sends it to Java in the model and requests the corresponding process. When Java in the model returns the processing result by strings, the control changes the format for the view using HTML. The procedure is elaborated as follows:

1) To show the assignment list in the view, JSP in the control receives the list with strings in the two dimensional array, changes them into the table format in HTML, and sends them to JavaScript in the view.
2) To show the selected assignment in the view, JSP receives the details with strings, changes them into the table in HTML, and sends it to JavaScript.
3) To mark the answers from the student, JSP receives them from JavaScript in the view and sends them to Java in the model. After completing the marking in the model, JSP receives the marking results from Java in the model, changes them into the table format in HTML, and sends it to JavaScript in the view.

IV. Implementation of JPLAS by Proposal

In this section, we describe the implementation of JPLAS by following the proposed software architecture. In this paper, only the functions which allow students to use JPLAS are implemented. We registered the assignments into the server database using SQL commands directly. The implementation of the functions for teachers will be explored in future works.

A. Overview of Implementation

Figure 7 exhibits the flow chart of data communication between the student, the user interface on the browser, the server, and the database. In this figure, 1) is executed by JavaScript after the browser receives the initial HTML file ”index.html” from the server. 2) is executed by JavaScript when a problem is chosen. Detailed data are provided by the server. 3) is executed by JavaScript when an assignment is submitted by the student. 4) and 5) are executed in the server by Java via JSP.

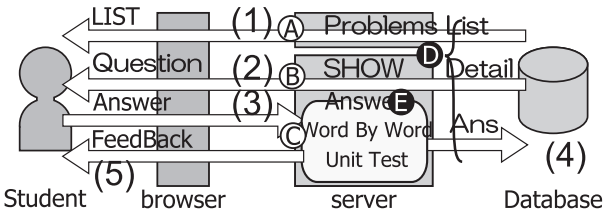

Fig. 7. Data communication flow chart in JPLAS.

In Figure 7, the functions for the assignment list display in A) are implemented in ”list.jsp” by JSP, the ones for the assignment display are in ”question.jsp”, and the ones for marking are in ”answer.jsp”. The functions for the database access in D) and for marking in E) are implemented in Java.

B. Implementation of Database Access

Figure 8 shows the set of classes to provide the database access functions corresponding to D) in Figure 7. In this figure, ”Db.class” represents the parent class for the database access functions. ”TableDb.class” gives the template to implement each table class. ”UserTableDb.class” implements the table containing the information about the students, ”QuestionTableDb.class” processes the table containing the assignments, and ”AnswerTableDb.class” does the table containing the answers from the students.

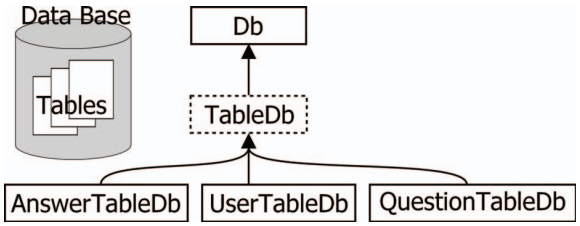

Fig. 8. Classes for database access.

1) Implementation of the marking function: Figure 9 shows the set of classes to provide the marking functions corresponding to E) in Figure 7. Each of the three marking functions in JPLAS was implemented by a class that adopts the abstract class named “Mark.class”. The *word matching test* was implemented in ”BlankMark.class”, the *compiling test* was in ”CompileMark.class”, and the *unit test* was in ”TestMark.class”. In “Mark.class”, the method to judge the feasibility of the corresponding test was in ”canMark ()”, the method to automatically select the next test was in ”setNext ()”. The applying order of the three tests is *word matching test* , *unit test*, and *compiling test*.

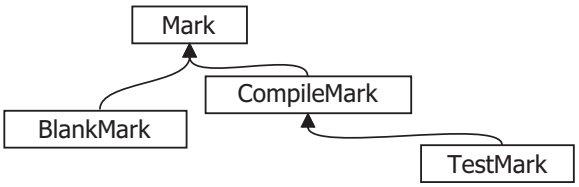

Fig. 9. Classes for marking.

V. Evaluation

In this section, we evaluate the proposed software architecture and implementation for JPLAS.

A. Number of Program Files in Implementation

First, we compare the number of program files between the implementation in the previous section and the original implementation of JPLAS. Table II shows the number of files in Java, JSP, CSS, JavaScript, and HTML respectively. More specifically, the numbers of files for database access functions in Java and JSP are also shown. To make accurate comparisons, we count the number of files only for the student functions in the previous implementation. This table shows that the number of required files is reduced by 70% of our proposal in this paper. Only one file is necessary for the database functions in the proposal.

TABLE II
Number of program files in two implementations of JPLAS.

file extension	original	proposal
java	81	21
database access	7	1
jsp	61	12
database access	16	0
css	9	6
js	40	38
html	122	11

B. Number of Program Files for New Functions

Next, we evaluate the number of additional program files required to implement two new functions into JPLAS.

1) Coding Rule Learning Function: As the first function, we evaluate the number of additional program files when we implemented the coding rule learning function [9]. This function will examine whether the code written by the student follows the coding rules or not by using open source software *Checkstyle* [9] and *PMD* [10]. To improve the readability and quality of the code, following the coding rules to write a code is an effective method.

2) Offline Answering Function: Then, as the second one, we evaluate the number of additional program files when we implemented the offline answering function [10]. This function allows the students to answer the assignments of JPLAS using the browser even when they cannot connect to the Internet.

3) Number of Additional Files for Two Functions: Table III shows the number of additional program files required to implement the two functions in JPLAS. In both functions, no new program files are necessary for the database access. The implementation of the offline answering function need only five JavaScript files additionally. However, the implementation of the coding rule learning function needs a number of additional program files. In future works, we will analyze the reason and consider the improvements of the proposed software architecture that can minimize the number of additional program files.

TABLE III
Number of additional program files for two new functions.

file extension	readable code	offline answering
java	1	0
jsp	10	0
css	17	0
js	31	5
html	3	0

VI. Related Works

In this section, we show our survey of Web application software structures using Java in a Web server.

In [11], the IBM Knowledge Center introduced the *Struts* framework and the model-view-controller design pattern. They show two JSP models, named *Model 1* and *Model 2* that can separate content generations or business logics from content presentations using HTML files. *Model 1* uses only JSP pages and Java beans codes. *Model 2* is the MVC model, where *Struts* helps to develop application software architectures on it.

In [12], Tani et al. proposed a method of implementing the front-end of a Web application using XSLT. It is implemented as a framework, that is, a collection of Servlet classes to become components of the target Web software where the overall logic is written with XSLT.

In [13], Nagao et al. proposed *Web Distributed MVC (WD-MVC)* architecture for realizing real-time Web applications. *WD-MVC* uses Ajax to convey messages from the Web server to the browsers asynchronously.

In [14], Ree et al. proposed *EcoFW* for end-user-initiative development projects. *EcoFW* focuses on the controller-configuration in frameworks such as *Struts*, whitch not only separates the view and the model, but realizes the view by components using Ajax and the JSON-format.

VII. Conclusion

In this paper, we proposed the software architecture for Java Programming Learning Assistant System (JPLAS) that can avoid redundancy in the codes and follows the MVC model using Java for the model (M), JavaScript/CSS for the view (V), and JSP for the controller (C). For the evaluation, we implement JPLAS by this architecture, compare the number of program files with the previous implementation and examine the number of additional files to implement new functions in

JPLAS. In future works, we will improve the architecture to minimize the number of additional program files and apply it to implement new functions in JPLAS.

ACKNOWLEDGMENTS

This work is partially supported by JSPS KAKENHI (16K00127).

REFERENCES

[1] N. Funabiki, Y. Matsushima, T. Nakanishi, and N. Amano, "A Java programming learning assistant system using test-driven development method," IAENG Int. J. Computer Science, vol. 40, no.1, pp. 38-46, Feb. 2013.
[2] Tana, N. Funabiki, and N. Ishihara, "A proposal of graph-based blank element selection algorithm for Java programming learning with fill-in-blank problems," Proc. IMECS 2015, pp. 448-453, March 2015.
[3] N. Funabiki, Tana, K. K. Zaw, N. Ishihara, and W.-C. Kao, "Analysis of fill-in-blank problem solutions and extensions of blank element selection algorithm for Java programming learning assistant system," to appear in Proc. WCECS 2016, Oct. 2016.
[4] N. Ishihara, N. Funabiki, and W.-C. Kao, "A proposal of statement fill-in-blank problem using program dependence graph in Java programming learning assistant system," Inform. Eng. Express, vol. 1, no. 3, pp. 19-28, Sep. 2015.
[5] Y. Higo, and N. Yoshida, "An introduction to code clone refactoring," JSSST, Computer Software, vol. 28, no. 4, pp. 43-56, Dec. 2011.
[6] K. Beck, *Test-driven development: by example*, Addison-Wesley, 2002.
[7] jQuery, http://jquery.com/.
[8] SkyBlue, https://stanko.github.io/skyblue/.
[9] N. Funabiki, T. Ogawa, N. Ishihara, M. Kuribayashi, and W.-C. Kao, "A proposal of coding rule learning function in Java programming learning assistant system," Proc. VENOA-2016, pp. 561-566, July 2016.
[10] N. Funabiki, H. Masaoka, N. Ishihara, I-W. Lai, and W.-C. Kao, "Offline answering function for fill-in-blank problems in Java programming learning assistant system," Proc. ICCE-TW 2016, pp. 324-325, May 2016.
[11] Struts framework and model-view-controller design pattern, http://www.ibm.com/support/knowledgecenter/SSRTLW_6.0.1/com.ibm.etools.struts.doc/topics/cstrdoc001.html.
[12] Y. Tani, N. Mitsuda, and T. Ajisaka, "A modular method and framework for Web application development using XSLT," IPSJ Tech. Report, 2003-SE-144, pp. 131-138, March 2004.
[13] T. Nagao, Y. Tsuchiya, S. Morimoto, and Y. Chubachi, "Realtime distributed MVC architecture using Ajax," Trans. IPSJ-PRO, vol. 48, pp. 200-200, June 2007.
[14] S. Ree and C. Takeshi, "Web application framework for end-user-initiative development," Proc. FIT2011, vol. 1, pp. 271-274, Sept. 2011.