# MVC Architecture Driven Restructuring to Achieve Client-Side Web Page Composition

Jaewon Oh[*], Woo Hyun Ahn[§], and Taegong Kim[#]
[*] *School of Computer Science and Information Engineering, The Catholic University of Korea, Korea,* jwoh@catholic.ac.kr
[§] *Department of Computer Science, Kwangwoon University, Korea,* whahn@kw.ac.kr, corresponding author
[#] *Department of Computer Engineering, Inje University, Korea,* sun@inje.ac.kr

*Abstract*—**This paper presents a restructuring approach to relocating web page composition from servers to browsers for Java web applications. The objective is to reduce redundant manipulation and transfer of code/data that are shared by web pages. The reduction is carried out through a restructuring algorithm, effectively keeping consistency between source and target applications from the perspective of the model–view–controller (MVC) architecture, because the problem requires the target application to preserve the observable behavior of its source application. Case studies show that our restructuring tool can efficiently support the restructuring process.**

***Keywords—restructuring; model–view–controller (MVC); single-page application; template-based web application; web engineering; software engineering***

## I. INTRODUCTION

In a web application, web pages usually share contents like headers, footers, and menus. A template is used to maintain such common contents in one place, which are separate from page-specific contents. A template-based web application (TWA) uses such a template to dynamically generate web pages, combining page-specific contents with the template. For example, in Fig. 1, the three pages are generated using a template and specific contents for each page.

TWAs basically interact with users, using the multi-page application model [12], where when a user issues a request, a web browser downloads and displays a new page. As shown in Fig. 2a, one of the most important things to note about TWAs is that the server combines and sends the template together with the page-specific contents upon each request. Therefore, the template is redundantly transferred and rendered with a full-page refresh for each request.

One of the solutions to the above duplication problem is to move page composition from servers to browsers. Whenever users click a hyperlink or submit a form, web browsers receive the differences between the current page and a new page, and then partially update the current page with the differences, without a full-page refresh. As shown in Fig. 2b, each of requests *Req1′* and *Req2′* causes a partial-page refresh with its associated differences. Page *P1* is physically a single page but can have different uniform resource locators (URLs), depending on its states (i.e., the included contents). Such web applications are called single-page applications (SPAs).

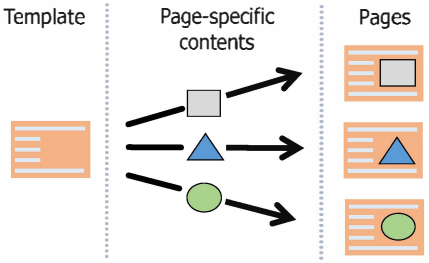
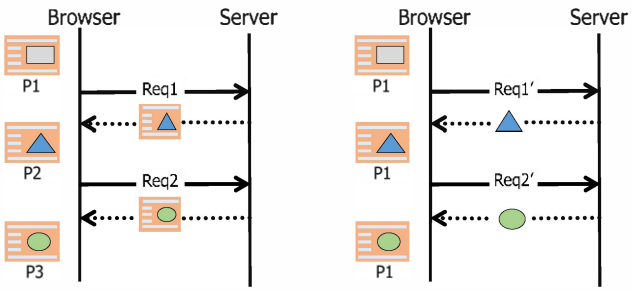



Figure 1. Template-based web application
In Java web applications, *<jsp:include>* standard actions are usually used to fill in placeholders of a template with page-specific contents.




(a) Sever-side composition (b) Client-side composition

Figure 2. Web page composition

The model–view–controller (MVC) [14] architecture is commonly used in web application development. Thus, it is natural to consider the three components and their interactions as a basic unit of analysis and modification when restructuring or reengineering multi-page applications into single-page applications. However, these architectural components have not been sufficiently discussed in most previous works about the transformation problem.

The transformation basically requires its precondition to be satisfied before it is carried out. In particular, reduction of the duplicated code/data sometimes induces inconsistency between source and target applications: the observable behavior of the target application can be different from that of its source application. However, most previous research on the transformation problem does not specify circumstances where the transformation approaches cannot be applied.

It is, therefore, necessary to specify the precondition and steps of the transformation in terms of the architectural

978-1-4673-9904-3/16/$31.00 

components of the MVC pattern. This paper proposes a way to decide whether a given full-page request can be restructured into a partial-page request, without changing the observable behavior of the source application, from the perspective of the MVC architecture. A restructuring algorithm is also presented to describe restructuring steps, as well as the restructuring precondition. We present the overall picture of the restructuring process with the execution of the algorithm as one activity in it. In addition, this paper describes the important issues that we encountered when implementing the restructuring algorithm. Moreover, we present the results of case studies showing that when our tool is used, the effort required for the restructuring is reduced, and the performance of web applications is improved.

## II. Problem and Background

### A. Problem

The restructuring problem addressed in this paper can be considered low-level restructuring because the user interface (UI) structure, the page navigation model (i.e., the network of links between pages), the data model, and the functionality of the source application are not changed. The target application has the same look and feel as its source application, but it improves performance in the response time and the amount of transferred data over the network, and provides asynchronous communication capability between servers and browsers. Moreover, coupling between templates and page-specific contents is reduced through the restructuring, and so, maintenance (e.g., the addition of new page-specific contents for new requirements) can be performed more easily. The rationale for choosing low-level restructuring is to seamlessly restructure the multi-page architecture of an input application into the single page architecture without confusing users.

We assume that the input TWAs were developed using the Java Server Pages (JSP) model 2 architecture [18] which adopts the MVC pattern. The assumption is reasonable, because the JSP model 2 architecture was mentioned in the early specifications of JSP and gained acceptance in the industry [11].

There are many approaches to the problem of restructuring or reengineering multi-page applications into SPAs. However, it is hard to find research that deals with the problem from the perspective of the MVC pattern, which is a popular architectural pattern for web application development. An earlier version of this paper [23] explicitly considered modification of models, views, controllers, and their interactions. However, the previous research had some limitations. The precondition of the restructuring was not described. An algorithm was not shown for automating the restructuring, so it is not easy to implement the approach. Experimental results were not shown regarding the restructuring process.

In the present paper, we improve our previous study [23] and propose a restructuring algorithm that considers consistency checking between input and output applications from the perspective of the MVC pattern. Moreover, we present the results of case studies in terms of process quality.

### B. Background

This subsection concisely describes our previous work [23] to help one understand the idea of the present paper more clearly. A Java web application basically consists of JSP, servlet, and JavaBean objects, which are mapped to views, controllers, and models, respectively, from the perspective of the MVC architecture. A web template has been used to provide users with a consistent view [9] and to reduce the code in web applications. The template creates and manages shared components, such as menus. When a user makes a request for a page, the request is sent to a servlet, which executes its business logic. The execution's result is then included in the template to produce the page, which is sent to the browser. JavaBeans objects are accessed for manipulation of business data by servlets and views during the request processing. Fig. 3a shows the dynamic architecture to process user requests.

The architecture has a problem in that the template is downloaded and rendered with a full-page refresh on each user request. Thus, our previous work [23] proposed an approach to restructuring the TWA towards client-side web page composition to solve the problem. The main idea is to download the template on first access to an application, and then, if possible, bypass downloading the template for subsequent accesses. In addition, user requests are transformed from normal hypertext transfer protocol (HTTP) requests to asynchronous JavaScript and XML (AJAX) [5] requests to enable a partial update in browsers. Full pages are completely composed in browsers with the help of JavaScript code, which updates the current Document Object Model (DOM) tree while replacing old page-specific parts with the downloaded parts. Fig. 3b shows the dynamic architecture to process user requests. Thus, the output application is an SPA.

Our previous work [23] also supports backward navigation and bookmarkability to enable accessing restructured applications in the same way as classic web applications. When a user clicks the back button of a browser in an SPA, the browser, by default, does not go back to the previous state of the SPA, but to a page visited before the SPA (see Fig. 4a). To solve the problem, our previous work [23] applied the hypertext markup language 5 (HTML5) history application programming interface [13], which stores current states in the browser history stack upon user requests and retrieves the states when the back button is used (see Fig. 4b). An SPA has different states but a single URL, by default. Thus, our previous work also proposed a method for assigning a unique request URL to each state in order to bookmark and revisit the state.

## III. Restructuring Template-Based Applications into Single-Page Applications

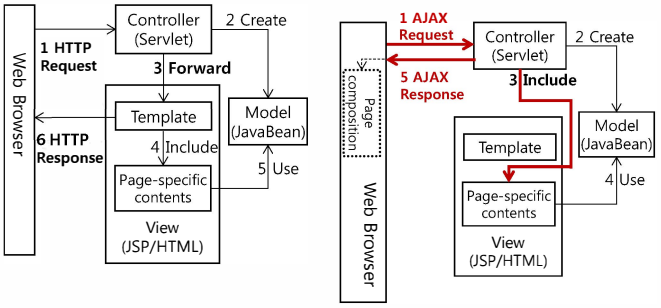
### A. Restructuring Process

In this subsection, we present the overall picture of our restructuring process to modify TWAs into SPAs (see Fig. 5). The steps are briefly described below.

First, an engineer executes and understands a source application from the user's point of view. Then, she collects links and forms issuing normal HTTP requests that can be

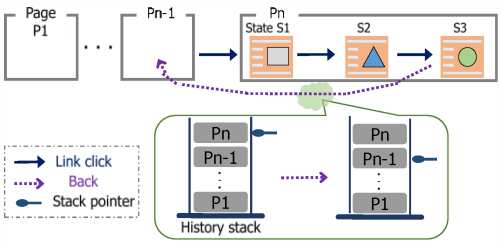
restructured. For each request, the engineer can extract a dynamic architecture, which can be represented as a collaboration diagram [2] similar to the diagram in Fig. 3a, to show collaboration among web components. The architecture can be used effectively in the validation step of the restructured application. Extraction of the architecture can be achieved using the filter and wrapper components provided in Java web development technology. The extraction step is not described in detail because of limited space. Now, the engineer understands the functionality and the architecture of the source application, and thus, a restructuring algorithm is executed to restructure the input application into an SPA. Validation of the output application is conducted after the restructuring is finished. When problems are found, including missed and incorrect transforms, manual transformation can be performed.

Note that this paper focuses on the architecture transformation and its support tool. Thus, we have developed an Eclipse plugin to support the "Execute restructuring algorithm" step of Fig. 5, which is described in the following sections in detail.

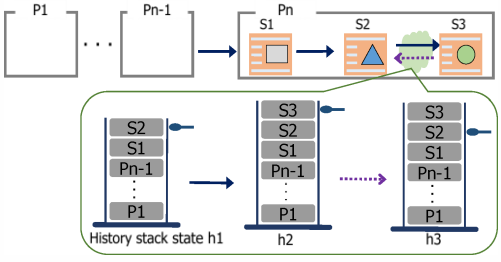



(a) Legacy architecture (b) Restructured architecture

Figure 3. Architecture transformation for client-side web page composition (adapted from Oh et al. [23])




(a) An example illustrating the problem




(b) An example illustrating Oh et al.'s solution [23]

Figure 4. Backward navigation problem and Oh et al.'s solution [23]

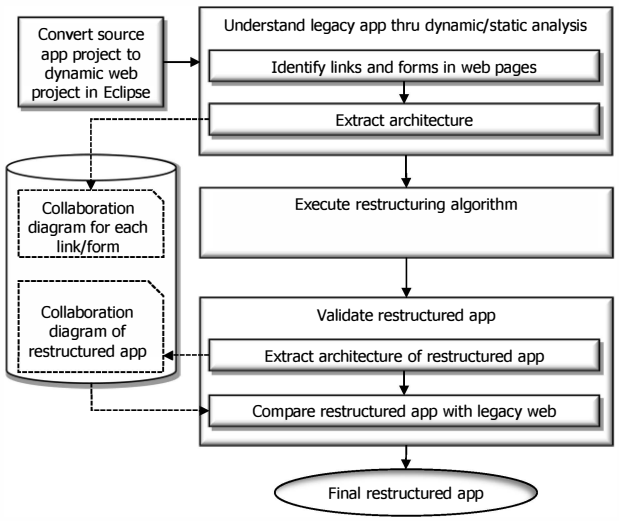



Figure 5. The overall resturcturing process

## B. *Restructuring Algorithm*

Fig. 6 shows how we transform a TWA into an SPA from an abstract point of view. First, the TWA is parsed to collect a list of full-page requests, each of which is then analyzed and modified, if possible, to a partial-page request by using *transformRequest()* (see lines 1-15). If the transformed request brings the generated SPA to an inconsistent state, the original request is not transformed (see lines 5-7). Thus, the output application can be a multi-page application, of which each page runs as a single-page application. Such inconsistent states can occur in three situations from the perspective of the MVC architecture (see lines 17-27).

The first case is where a transformed request generates a view that does not exist in the TWA (see line 20). For instance, suppose we have a full-page request *r* that changes the current template as well as the page-specific contents. If *r* is transformed into a partial-page request that downloads new page-specific contents, the transformed request produces an inconsistent view.

A template is not a static page but a dynamic page (see Fig. 7). Different template objects are instantiated from a template class, depending on the system/user state. For example, in the figure, template objects are replaced after a user logs on because their menus are different. When a user request needs a new template instance, an SPA should also download the new instance. One of the possible designs is to download the differences between the old and new template objects. However, the differences can be scattered throughout the template objects, so the implementation is not easy. Thus, we make a decision to not transform the original full-page request.

The second case is where the transformation of an original request induces improper situations: 1) an invalid controller is introduced into the SPA, or 2) a request is made to a different controller than we expect in the TWA (see lines 21-23).

```
1:  procedure transformRequest(Request r)
2:  // let r be the full-page request currently considered for restructuring
3:  // let Δ be page-specific contents of r
4:  // let url_Δ be a request URL for Δ
5:  if r's transformation causes an inconsistent state then //use isConsistent(r)
6:      return // r is not changed but used the same as in the input app
7:  end if
8:  change r into a partial-page request with url_Δ so as to only obtain Δ
9:  replace old page-specific contents with Δ
10: change current URL of browser into URL of the newly composed page
11: push the new URL into browser's history stack
12: if r uses web template t for response generation on server then
13:     modify r such that r bypasses t and generates Δ
14: end if
15: end procedure
16:
17: procedure bool isConsistent(Request r)
18: // let r_t be the transformed partial-page request of r
19: if one of the following occurs
20:   (i) r_t generates a view that does not exist in the input app
21:   (ii) r_t introduces a controller that does not exist in the input app
22:   (iii) r_t causes a request to be made to a different controller than expected in the input
23: app
24:   (iv) r_t generates a model that does not exist in the input app then
25:       return false
26: return true
27: end procedure
28:
29: // include following code in SPA to support backward and forward navigation of
30:  // pages composed in browsers
31: procedure onBackForwNavigationPerformed()
32: // precondition: triggered when user presses back or forward button to move to
33: // page that has been composed in browser
34: restore target page using the current entry en in browser's history stack
35: change current URL of browser to URL of en
36: end procedure
37:
38: //include following code in SPA to efficiently support full-page refresh
39: procedure onTemplateLoaded()
40: // precondition: triggered when template is loaded on browser
41: // let Δ_r and url_Δ have the same meanings as in procedure transformRequest()
42: request Δ with url_Δ without inducing full-page refresh
43: place Δ in web template
44: end procedure
```

Figure 6. Restructuring algorithm

First, a web browser executes client code in HTML, cascading style sheets (CSS), and JavaScript. Among them, JavaScript code plays the role of controllers on the client side. One of the important things to note about client code is that when page-specific contents are replaced, the side effect of the execution of JavaScript code from the old page-specific contents still remains, unlike HTML and CSS code. On the other hand, a full-page request unloads all the client code of the old page. Thus, analysis is needed to identify whether old JavaScript code lets the SPA run differently from the TWA.

The specific scenario of changing the observable behavior of a TWA is as follows: For instance, suppose that a page-specific part of some page in a TWA has JavaScript code to download images as soon as a user scrolls to the bottom of the page (e.g., through AJAX requests within the *window.onScroll()* event handler), in order to reduce the page load time. If the user moves to a new physical page, such event handling code is removed automatically owing to the page unload. However, in the restructured SPA, the code removal does not occur automatically, because entering a new state does not cause a full-page refresh. It means that an invalid controller exists in the SPA (line 21).

Second, an SPA should preserve the page navigation model (including the backward/forward navigation and refresh) of its input TWA. For example, suppose that the TWA's page $p_t$ is restructured into SPA page $p_s$, so that $p_s$ is the same as $p_t$. If the two pages are refreshed, the new pages should also be the same. To achieve this, each state of an SPA has a unique URL, like each page of a TWA. A URL of such a state needs to include the information to identify the state (e.g., a request URL for obtaining its page-specific contents). Thus, when a page is refreshed, page-specific contents can be obtained together with the template (see Fig. 8). There is one important thing to note when restructuring a full-page request into a partial-page request using AJAX: when an initial request is redirected (through an HTTP redirect request made from the server to the web browser) to another component *c*, the redirected URL (i.e., the request URL of *c*), by default, cannot be obtained by an SPA. On the other hand, the redirected URL is shown in the address bar of a web browser with a TWA. Thus, when we try to refresh a page of an SPA that includes the response of an HTTP redirect request as page-specific contents, the SPA runs differently from its input TWA. It means that the SPA sends the request to an improper controller (lines 22-23).

The third case is where a transformed request can generate a model that does not exist in the TWA (see line 24). For instance, the transformation of original requests that consistently update the server state can permit the generated SPA to have invalid data. A more specific scenario is as follows: suppose that, in the TWA, we have a page with a payment form that cannot be submitted with the same data more than once. However, if its transformed form is allowed to be submitted with the same data twice, the transformation adversely incurs a double payment, which is an instance of the double submit problem [1, 8, 19]. An even more specific example is shown in the next section.

On the other hand, if the transformation of an original request induces a consistent state, the transformation is

performed (see lines 8-15) such that the output application works as follows: through the transformed request, a browser receives a new page-specific part (line 8) and replaces the old part with the new part (line 9), without a full-page refresh. In order to transfer such a page difference from the server to the browser, a request URL ($url_{\Delta}$ in lines 4 and 8) identifies its page-specific part. In addition, a unique URL is assigned to the full page so users can bookmark and revisit the page (see lines 10 and 11, and Fig. 8). A controller is modified such that its interaction with the template is skipped and it connects to the view generating the page-specific contents (lines 12-14).

*onBackForwNavigationPerformed()* is newly added in the SPA in order to enable users to navigate web pages backwards and forwards (see lines 29-36). If a user presses back or forward buttons and the target page has been composed in the browser, we obtain (by using the history stack) the previous state, which is then inserted into the current page to restore the target page. Then, the URL of the browser changes to the URL of the target page. On the other hand, when a page is newly composed in the browser, information about the new page is pushed onto the history stack (see line 11).

*onTemplateLoaded()* is also added in the SPA in order to efficiently support full-page refreshes (see lines 38-44). A page is fully loaded when users enter SPAs or refresh the current page. Such full-page refreshes require downloading both the template and the page-specific parts. We use the hybrid approach (as shown in Fig. 8) between server-side and client-side web composition, in order to enable parallel downloads of web resources for the page. Thus, the template is first requested; the page-specific part is then requested immediately, without a full-page refresh, similar to the partial-page update steps (lines 8-9) of the transformed request.

## IV. IMPLEMENTATION

This section describes the Eclipse plugin that we have developed to restructure TWAs into SPAs. The implementation issues of the restructuring algorithm equipped within the tool are also described in detail.

### A. Restructuring Tool

A restructuring tool runs with original files in the dynamic web project of Eclipse in two steps. In step 1, code segments to be transformed are identified based on the precondition of our restructuring. The tool uses the Jericho HTML parser [7] to search JSP and HTML files for such code segments. The parser was chosen because JSP and HTML tags are recognized and can be also modified by using the parser. In step 2, the identified code snippets for restructuring are changed by using the restructuring algorithm.

### B. Implementation of Restructuring Algorithm

This subsection describes the important issues that we encountered when implementing the restructuring algorithm. One of the issues was to determine whether a transformed request brings the generated SPA to a valid state. The restructuring tool currently considers the following factors for such a determination.

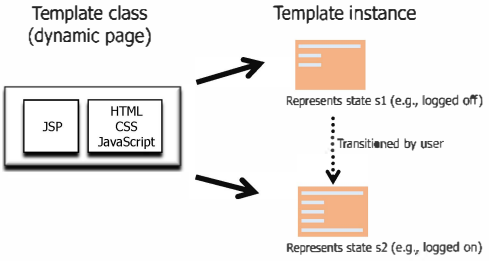



Figure 7. Template and its instantiation

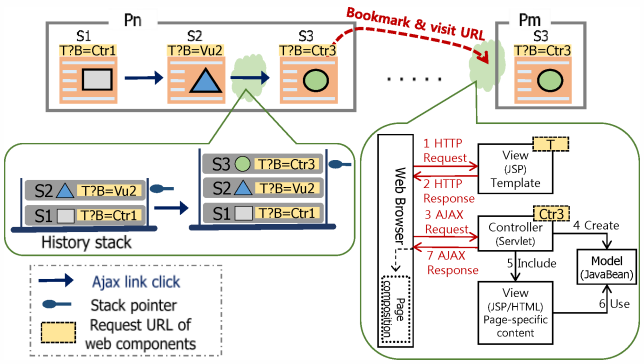



Figure 8. Solution for bookmarkability and page refresh

Suppose that in state *S3* of page *Pn*, a request URL for the page-specific part is *Ctr3*. In addition, let a request URL for the template be *T*. We can assign "*T?B=Ctr3*" to a full page including state *S3*.

First, the tool considers HTTP request methods, among which GET and POST are analyzed deliberately in the determination process, because the two methods are the most commonly used for web applications. HTTP GET requests are generally restructured in our tool because the GET method is designed to retrieve resources from servers, and the transformation of HTTP GET requests does not cause side effects.

On the other hand, POST requests can be transformed or not, depending on how applications handle the requests. Generally, HTTP POST requests are designed to update the server state, and can cause the double submit problem if the requests are not processed carefully. Thus, two solutions to the problem have been proposed in the literature: the post–redirect–get (PRG) pattern [1, 8, 20] and the synchronous token mechanism [1, 20]. When an original request of a TWA uses the POST method with the synchronous token mechanism, the transformation of the request does not induce the double submit problem. However, the transformation of a POST request with PRG pattern support can cause the problem because the redirected address cannot be obtained. Thus, the decision to transform POST requests requires more static and dynamic analysis than for GET requests. The decision might also need the intervention of engineers.

Second, the tool considers the context in which a request is issued. Suppose we have a POST request that causes the

double submit problem if the request is transformed and issued in the browser's main window. However, when the request is issued in window *w* (other than the browser's main window), and *w* disappears after processing the request, the request cannot be made with the same data twice. Thus, the request does not cause the problem. It can be seen that a request can be transformed or not, depending the context of the request.

Third, to transform and test the request can be considered a criterion for determining transformation of a request. This approach can be effective, but not cost-effective. The essential precondition of refactoring/restructuring is to have solid tests [24]. Thus, the approach can be useful and efficient when test cases are prepared properly.

Another implementation issue we encountered is the detection and preprocessing of applications that break the principle of HTTP request methods. For example, clicking <*a*> elements is normally issued as HTTP GET requests to retrieve information from servers. However, we can find applications that have some web pages containing <*a*> elements where clicking causes server state changes. Thus, preprocessing is performed to statically and dynamically analyze hyperlinks and change the method types appropriately, before executing the restructuring algorithm.

Another implementation issue is to identify or build URLs of page-specific contents. When a request URL for a page-specific part is sent to a template as a request parameter, the values of the parameter can be simply used to build such URLs. On the other hand, an application can use logical identifiers other than request URLs to designate page-specific parts. The identifiers can then be mapped to request URLs on the server side. In this case, such a mapping table should be collected before/during execution of the transformation algorithm.

## V. CASE STUDIES

In this section, the results of case studies are described in terms of process quality and product quality.

### A. Process Quality

In this subsection, we present the results of case studies from the perspective of process quality. We performed experiments on three TWAs: an online book publisher [10], a web community to share information on the rearing of cats, and another web community to help raise babies. For each TWA, restructuring with and without tool support were performed. Table I shows the size of the applications in terms of the number of files, the lines of code (LOC), the number of hyperlinks including forms (total#), and the number of the hyperlinks to be transformed (transformed#).

This paper uses four well-known metrics (precision, recall, accuracy, and effort) to evaluate the results of the experiments. The precision metric measures how many hyperlinks transformed by our tool (or the experimenters) are hyperlinks to actually be transformed. Recall represents how many hyperlinks to be transformed are correctly transformed by our tool. Accuracy measures how accurately our tool transforms hyperlinks to be transformed. Effort measures how much time is spent to complete the restructuring.

The experimenters in the three case studies, having similar capabilities and experience with web application development, were already acquainted with our restructuring approach before the experiments. The experimenters who manually transformed the applications used the same JavaScript library provided in our tool.

Fig. 9 shows the results. In all the case studies, when our tool was used, the effort (person-hours) for the restructuring was reduced. All the experimenters accurately transformed the three applications, so the values for precision, recall, and accuracy are all 100%. In addition, the tool-supported restructuring of the online book publisher and the community for cat care were more efficient than that of the community for baby care. One of the reasons is as follows: the community for baby care has complex interactions among web components, some of which play the role of both a controller and a view. Thus, the transformation required deep understanding of the community for baby care. The other reason is that the experimenter using our tool tried to refactor the dynamic architecture of the community for baby care to remove unnecessary interaction, unlike the experimenter without tool support.

### B. Product Quality

To validate our approach from the perspective of product quality, we measured the performance (i.e., the response time and the amount of transferred data over the network) of web applications on mobile devices as well as desktop computers.

We chose as the case study, the TWA for the online book publisher [10]. This application can be considered a simplified version of certain online store applications [3, 11], often used as benchmarks for web applications. Another reason to choose this TWA is that the application has diverse web pages in terms of receivers of HTTP requests, the type of page-specific contents, etc., as seen in Table II. For example, the *Find* page allows users to search for books by keyword. When a user enters a keyword in a text field within a form and submits the form, the GET request is sent to a controller, which sends queries to a database to extract relevant information.

The TWA being restructured is named *Tapp*, which is transformed into *Sapp* through our approach. Our experiments were conducted on two desktop computers (i.e., a server and a client) with a 3.3 GHz Intel i5-2500 processor, 4GB DDR3 RAM, and the Windows 7 32-bit operating system. Data were stored in a Cubrid 2008 R4.1 database. Tomcat 7.0 and Chrome 35.0.1916 were used as a web container and a web browser, respectively.

The performance of the template download and web page composition needs to be measured to show the efficiency of the client-side page composition proposed in this paper. Data caching in browsers influences web application performance. Thus, we performed experiments in two situations: one is when web pages are first accessed, so cache misses occur. The other is when web pages that have already been visited are accessed, so cache hits occur.

TABLE I. APPLICATIONS UNDER CASE STUDY

| | Online book publisher | Online community for cat care | Online community for baby care |
|---|---|---|---|
| Servlet (# of files, LOC) | (11, 668) | (9, 571) | (26, 1944) |
| JSP | (18, 461) | (18, 715) | (33, 965) |
| HTML | (3, 16) | (0, 0) | (0, 0) |
| JavaScript | (1, 34) | (1, 82) | (0, 0) |
| JavaBean | (7, 380) | (1, 87) | (11, 740) |
| Java, CSS, XML | (2, 125) | (4, 807) | (5, 384) |
| Total (# of files, LOC) | (42, 1684) | (33, 2262) | (75, 4033) |
| Hyperlink and form (total#, transformed#) | (24, 15) | (18, 9) | (41, 21) |

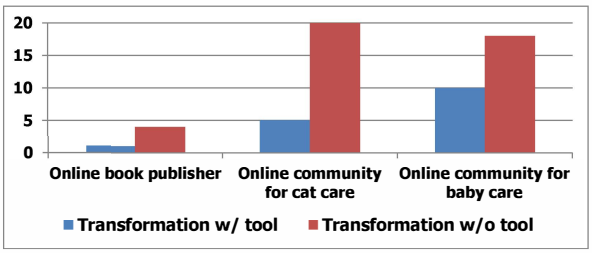



Figure 9. Results of case studies in terms of effort (person-hours)

Fig. 10 shows the performance results when web pages are first accessed. We can see the performance of the template download by accessing the *Main* page, which is the first page that users visit and only shows the template. *Sapp* is slower than *Tapp* because *Sapp* downloads and processes not only the template but also the JavaScript code (see page *Main* in Fig. 10b) to enable AJAX requests. After loading the template into the browser, however, *Sapp* is faster than *Tapp* when accessing pages allowing client-side page composition (i.e., *Intro*, *BookList*, *RevuFm*, and *Find*).

The *RevuWr* page is used to write book reviews by issuing a form POST request. *Sapp* refreshes a full page, because if the request for the page is transformed, the transformed request causes the double submit problem. The full page of *Sapp* is generated with two steps, as shown in Fig. 8: the template download and client-side page composition. When compared to the *Main* page, the performance difference between *Sapp* and *Tapp* is smaller. This is because the download of web resources composing the template (e.g., images with a total size of 240 KB, JavaScript code, and CSS code) occurs concurrently with the execution of transactions by the server.

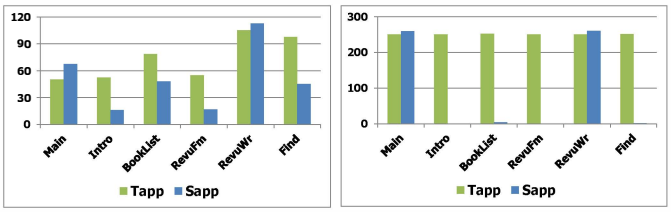
Fig. 11 shows the results when web pages that have already been visited are accessed. Note that the static resources composing the pages are cached in the browser, and so the amount of transferred data is reduced, compared to the first experiment, as seen in Fig. 10b and Fig. 11b. Thus, the response time is also reduced, as can be seen in Fig. 10a and Fig. 11a. *Sapp* is much faster when accessing the *Intro* page because the page-specific part is static HTML code (see Table II), which is already cached in the browser.

From the two experiments, we can see that *Sapp* improves the performance of *Tapp* when accessing the pages that cause partial-page refreshes, regardless of data caching in browsers.

Access to the web from mobile devices like smartphones is becoming more popular. Users want to efficiently navigate web pages on mobile devices, as well as on desktop computers. Thus, we also measured the performance of *Sapp* and *Tapp* on a smartphone with a 1.2GHz processor, 1GB RAM, and the Android 4.1.2 Jelly Bean operating system, in order to determine performance improvement on mobile devices. Fig. 12 shows the response time of the two applications when a 3G network is used. It can be seen that much improvement can be achieved using our restructuring approach. One of the reasons is that the full-page refresh requires more CPU time to display pages. The amount of transferred data is the same as the experiment in the desktop environment. Thus, users of *Sapp* on mobile devices can reduce costs if they pay based on the amount of transferred data.

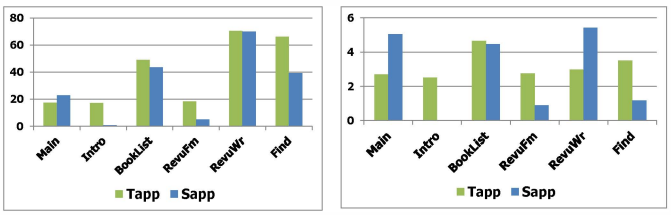
TABLE II. WEB PAGE DIVERSITY FOR THE ONLINE BOOK PUBLISHER

| Page | Description | HTTP request receiver | Page-specific contents | Database transaction use | HTTP method | Form |
|---|---|---|---|---|---|---|
| Main | main page; show only template | template | empty | N | GET | n/a |
| Intro | describe company | template | static HTML | N | GET | n/a |
| BookList | show info on books | controller | dynamic HTML | Y | GET | n/a |
| RevuFm | load form to write review | template | dynamic HTML | N | GET | download |
| RevuWr | write book review | controller | dynamic HTML | Y | POST | submission |
| Find | search books by keyword | controller | dynamic HTML | Y | GET | submission |



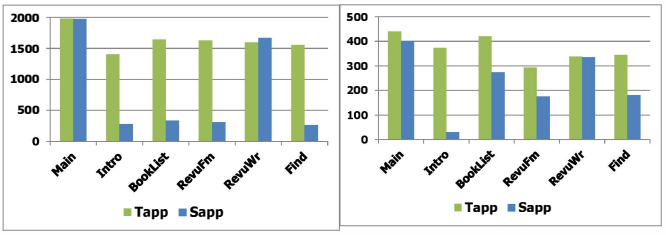

(a) Response time (ms) (b) Transferred data (KB)

Figure 10. Performance of Tapp and Sapp for web page composition on the first access to each page.




(a) Response time (ms) (b) Transferred data (KB)

Figure 11. Performance of Tapp and Sapp for web page composition on access to each page already visited.




(a) On the first access to each page (b) On the second access to each page

Figure 12. Response time (ms) of Tapp and Sapp on mobile devices

## VI. DISCUSSION

Even though our approach was applied to only three case studies, we believe the approach can be scalable to cover more applications. One of the reasons is that JSP model 2 web applications considered for restructuring in this paper have been adopted and are largely used in the industry [11]. Second, the three applications are representative of TWAs. The online book publisher is a simplified version of online book stores used as benchmarks in previous studies [3, 11, 23]. The others are simplified versions of typical web community sites for sharing information about specific themes. Third, we do not require the restructured SPA to use specific libraries or plugins. In addition, we will conduct more case studies including open source software to improve our approach and tool.

Our approach currently assumes that the difference between consecutive pages of a TWA is localized spatially in the full page. Thus, the changed part is encapsulated into one cohesive module using a <*div*> or <*span*> element. To relieve the limitation, the browser-side templating (BST) [6] approach can be adopted because it allows placeholders of a template to be scattered in a template using context variables.

One research need we found when conducting the case studies is the extraction of a dynamic architecture model that shows collaboration among web components (such as servlets, JSP pages, and JavaBeans) to complete a user request. The model can be a graph, similar to the graph shown in Fig. 3a, in which nodes are web components, and edges are relations between the components (such as *forward*, *include*, and *redirect* actions). Such graphs for TWAs and SPAs can be considered and used as abstract models to understand and validate TWAs and SPAs. The extraction of the architecture can be achieved using the filter and wrapper component provided in Java web development technology. The extraction step is not described in detail in this paper for want of space.

We think that the restructuring algorithm can be applied to web applications implemented using other MVC-based development frameworks. That is because, although the current study focuses on the JSP model 2 architecture, we basically use architectural components of the MVC pattern and web standards other than architectural components of the JSP model 2 architecture for the description of how to carry out our restructuring.

The restructuring algorithm can be implemented using existing libraries (such as PJAX [15]) other than web standards such as HTML5. Then, engineers can concentrate more on the application domain from the perspective of the MVC pattern, without putting much effort into implementing low-level functionality, such as backward navigation.

## VII. RELATED WORK

This section analyzes state-of-the-art web application restructuring/reengineering towards client-side web page composition from the perspective of the MVC pattern. There are two main approaches to such problems [23]. First, client code analysis focuses mainly on the refactoring of client code written in HTML, CSS, JavaScript, etc. Second, server code analysis focuses on the transformation of server components such as servlets, JSP pages, and JavaBeans.

Several studies [12, 16, 17] that are classified as client code analysis focused mainly on UI transformation, which modifies the UI structure (e.g., UI layout) and the page navigation model for input applications. From the perspective of the MVC pattern, these studies mainly considered the modification of views. The interaction between controllers and views, however, should also be refactored to transfer the difference between consecutive pages from servers to browsers.

The studies of server code analysis mainly considered transforming controllers, models, and the interactions among models, views, and controllers. One study by Ying and James considered the ajaxification of HTML forms [22]. Another study [4] considered transformation of paginated web pages, which display a list of data, into a single page. From the perspective of the MVC pattern, these two studies considered the modification of special views and their associated controllers and models. Thus, it is insufficient to apply these studies to general TWAs.

Another study classified as server code analysis [21] considered the restructuring of general JSP-based applications into single-page applications. The study focused mainly on modifying the processing mechanism of user requests to enable partial-page refreshes with the help of a newly introduced front controller [1, 20]. The study also presented a process for the restructuring and the results of a case study in terms of process quality. The preservation of the page navigation model of a source application was not addressed in the study.

The server code analysis approach recently proposed by Oh et al. [23] differs from the above-mentioned previous works. First, the approach modifies request URLs of input applications to obtain software architecture with partial-page refreshes. Second, the approach supports backward navigation and bookmarkability to enable accessing restructured applications in the same way as classic web applications. However, an algorithm was not shown for automating the restructuring, so it is not easy to implement the approach. Experimental results were not shown regarding the restructuring process. That study did not specify circumstances where the transformation approaches cannot be applied.

To achieve partial-page refreshes, AJAX is adopted in our approach. BST can be considered as an alternative solution. BST is different from AJAX in that there exist template definition languages interpreted in the BST engines of browsers. The engine replaces placeholders in a template with

data transferred from a server. One of the strengths of BST is that the engine can generate pages even though placeholders are randomly scattered in a template. However, templates and models for page-specific contents should be prepared according to the template framework. Such preparation for the restructuring is expected to require much effort. Moreover, the issue of refactoring interactions among models, views, and controllers still remains.

## VIII. CONCLUSION

We proposed a technique to restructure template-based web applications into single-page applications. The objective is to reduce redundant manipulation and transfer of code/data shared by web pages.

The main contributions of this paper can be described as follows. First, we presented the overall picture of the restructuring process. Second, we proposed how to decide whether a full-page request can be restructured into a partial-page request from the perspective of the MVC architecture and the page navigation model. Third, we described the steps necessary to perform the restructuring through the restructuring algorithm in terms of the MVC architecture. Fourth, we discussed the issues of the tool that implements the restructuring algorithm, and showed that the tool can efficiently support the restructuring process.

Our emphasis on reducing redundant download and display of web pages is consistent with a growing need for responsive web applications on mobile devices. Users want to efficiently navigate web pages on mobile devices, as well as on desktop computers. However, mobile devices have less CPU power than desktop computers, so full-page refreshes in mobile browsers increase the response time. As shown in the case studies, our restructuring approach to client-side web page composition can be effective at reducing the response time in mobile browsers.

In our ongoing work, we will conduct more case studies to find weaknesses in our approach and tool in order to improve them. Automatic architecture extraction from source code will also be considered.

## ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF-2011-0023224 and No. 2011-0013781). This work was supported by the Catholic University of Korea, Research Fund, 2016.

## REFERENCES

[1] D. Alur, J. Crupi, and D. Malks, Core J2EE Pattern. Prentice Hall PTR, 2003.

[2] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide. Addison Wesley Longman Inc., 1999.

[3] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, “Performance comparison of middleware architectures for generating dynamic web content,” Proceedings of ACM/IFIP/USENIX 2003 International Conference on Middleware, Rio de Janeiro, 2003.

[4] J. Chu and T. Dean, "Automated migration of list based JSP web pages to AJAX," Proceedings of 8th IEEE International Working Conference on Source Code Analysis and Manipulation, Beijing, 2008.

[5] M. Eernisse, Build Your Own AJAX Web Applications. SitePoint Pty Ltd, 2006.

[6] F.J. Garcia-Izquierdo and R. Izquierdo, "Is the browser the side for templating?," Internet Computing, vol. 16, no. 1, pp. 61-68, 2012.

[7] Jericho HTML Parser. Available from http://jericho.htmlparser.net.

[8] M. Jouravlev. Redirect after Post. Available from http://www.theserverside.com/articles/article.tss?l=RedirectAfterPost.

[9] C. Kim and K. Shim, "Text: automatic template extraction from heterogeneous web pages," IEEE Transactions on Knowledge and Data Engineering, vol. 23, no. 4, pp. 612-626, 2011.

[10] Y. Kim, JSP & Servlet for Java Programmers. Hanbit Media Press, pp. 717-793, 2010, (in Korean).

[11] B. Kurniawan and J. Xue, "A comparative study of web application design models using the Java technologies," in Yu, J.X., Lin, X., Lu, H. and Zhang, Y. eds. Advanced Web Technologies and Applications, Springer Berlin Heidelberg, pp. 711-721, 2004.

[12] A. Mesbah and A. Van Deursen, "Migrating multi-page web applications to single-page AJAX interfaces," Proceedings of 11th European Conference on Software Maintenance and Reengineering, Amsterdam, 2007.

[13] M. Pilgrim. Dive into HTML5. Available from http://diveintohtml5.info.

[14] Y. Ping, K. Kontogiannis, and T.C. Lau, "Transforming legacy web applications to the MVC architecture," Proceedings of 8th Annual International Workshop on Software Technology and Engineering Practice, Amsterdam, 2003.

[15] PJAX. The PJAX library. Available from https://github.com/defunkt/jquery-pjax.

[16] R. Rodr?guez-Echeverr?a, J. Conejero, P. Clemente, J.C. Preciado, and F. S?nchez-Figueroa, "Modernization of legacy web applications into rich internet applications," in Harth, A. and Koch, N. eds. Current Trends in Web Engineering, Springer Berlin Heidelberg, pp. 236-250, 2012.

[17] G. Rossi, M. Urbieta, J. Ginzburg, D. Distante, and A. Garrido, "Refactoring to rich internet applications. A model-driven approach," Proceedings of 8th International Conference on Web Engineering, New York, 2008.

[18] G. Seshadri. Understanding JavaServer Pages Model 2 Architecture. Available from http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html.

[19] I. Vosloo and D.G. Kourie, "Server-centric web frameworks: An overview," ACM Computing Surveys, vol. 40, no. 2, pp. 4:1-4:33, 2008.

[20] P. Wang, Comparison of Four Popular Java Web Framework Implementations: Struts1. X, WebWork2. 2X, Tapestry4, JSF1. 2. Master's Thesis, University of Tampere, Finland, 2008.

[21] Q. Wang, Q. Liu, N. Li, and Y. Liu, "An automatic approach to reengineering common website with AJAX," Proceedings of 4th International Conference on Next Generation Web Services Practices, Seoul, 2008.

[22] M. Ying and M. James, "Refactoring traditional forms into AJAX-enabled forms," Proceedings of 18th Working Conference on Reverse Engineering, Limerick, 2011.

[23] J. Oh, W. H. Ahn, S. Jeong, J. Lim, and T. Kim, "Automated Transformation of Template-Based Web Applications into Single-Page Applications," Proceedings of IEEE 37th Annual Computer Software and Applications Conference, Kyoto, 2013.

[24] M. Fowler, Refactoring: improving the design of existing code. Pearson Education India, 2009.