

Visual Web Application Composition Using WebPads

Marcel Karam¹ Sergiu Dascalu² Rami Santina¹ Zeina Koteich¹ Rana Awada¹
¹*American University in Beirut, Lebanon* ²*University of Nevada, Reno, USA*
marcel.karam@aub.edu.lb *dascalus@cse.unr.edu*

Abstract

The task of engineering remote service-oriented web applications usually adds extra layers of complexity to the web engineering production process. The approach described in this paper allows developers who employ the Model View Workflow (MVWf) paradigm to visually wrap local and remote services and specify their dependencies using a concept we call WebPads. WebPads are implemented in our proposed framework as the visual constructs' building blocks of the remote service web development process. To evaluate our solution and assess its usability we conducted a couple of studies. Feedback from these studies shows the usefulness of our technique in reducing the complexity of remote service-oriented web application production as well as the worthiness of creating a visual front-end environment, VisualWebC, that supports and integrates our approach in open source MVWf-based web development. The potential use of WebPads in the context of software product line engineering is also discussed in the paper.

1. Introduction

Interconnected web systems that provide service-oriented functionalities have become the norm in today's landscape of web applications. Stock quotes, weather updates, credit cards payment, and currency converters – to name only a few – are *services* available on many different interconnected web systems. Emerging web applications seeking to provide, among other things, similar services, usually make use of shared deep annotation architectures that rely, in general, on three major “players”: (i) database owner, (ii) annotator, and (iii) querying party [1]. This approach depends heavily on the metadata construction methods used by the host applications and can have many drawbacks [2]. Other new approaches rely on loosely coupled solutions, such as Service-Oriented Architectures (SOA). While such models are convenient, they are not very suitable for remote service-oriented interactions [3].

Our approach to engineering remote service-oriented web applications, described in this paper, relies on the

ability of *wrapping* these web services and making them available to a web engineer in a way that allows the engineer to “peel off the services” and visually (and simply) integrate their iconic representations in the web application being built, during the development process. More specifically, we decided to integrate our wrapping approach into a *Model View Workflow (MVWf)*-based framework that allows the user to visually engineer the web application and its remote services. Our decision to use the *MVWf* paradigm [4, 5, 6] is based on our belief of being able to augment this paradigm with a powerful graphical interface that allows the developers to visually construct remote services and manage their dependencies as workflows. To evaluate our solution we implemented *VisualWebC* (short for *Visual Web Composition*), an *MVWf*-based environment that we used for our design and development experimentations. The concept and implementation of wrapping web services as *WebPads* in *VisualWebC* are extensions of the concept and implementation of *Pads* proposed by Ito and Tanaka [7]. A *Pad* wraps services of existing web applications and plugs them into a *desktop application* that is based on the Intelligent *Pad Architecture* [8]. As shown in [7], this approach has proved to be very useful in allowing programmers to create their own desktop applications services from remote web applications, thus saving time and effort during both development and maintenance.

To provide web developers with a way to quickly integrate remote web services in newly created web applications and take advantage of some of the benefits that are available when using pads for desktop applications, we extended the concept of *Pads* to *WebPads* in *VisualWebC* and thus provided software engineers with the ability to visually configure, manage, and integrate already available and tested remote web services in their new *web applications*. This approach to developing remote service-oriented web applications provides the compelling benefits of reducing the time, effort, and overall complexity involved in re-engineering these services. Furthermore, wrapping and using a remote service also reduces the maintenance and upgrading involved in having such service integrated in one's own web application, since it is provided as an “off-the-shelf” service.

The remaining of this paper is organized as follows: Section 2 examines related work, Section 3 describes our *VisualWebC* prototype and its main components, Section 4 further presents features incorporated in *VisualWebC* and discusses the advantages and the disadvantages of our solution, Section 5 presents two short studies conducted to evaluate our approach’s usability, Section 6 outlines possible uses of our web application development solution as an agile instrument and in support of software product line engineering, and Section 7 rounds up the paper with pointers to future work and several concluding remarks.

2. Related work

In [8] and [9], the authors present, in the context of engineering desktop applications, the *intelligent pad architecture*, in which each component is represented as a *pad*. A pad can be pasted onto another pad to define both a physical containment relationship and a functional linkage between the two pads. This linkage defines a parent-child relationship between pads, where no pad can have more than one parent pad. This link definition dictates the fact that the recursive pasting of pads onto one another results in a tree pad structure. Communications between the pads occur through standard messages such as “set”, “gimme” and “update”. In [7], Ito and Tanaka describe the tool they created for combining different web applications into a single desktop application. Their tool wraps parts of existing web applications and plugs them on a desktop application using the intelligent pad architecture. The functionalities of the pads use the Document Object Model (DOM) [10] to identify objects within HTML documents as well as the browser API to reference the components, manipulate their content, and trigger their associated methods. In [8], [11], and [12] the authors extend the work done in [7] to allow users to clip elements from existing applications and form cells on a spreadsheet, connect these cells using formulas, and clone these cells to provide the mechanism of handling multiple requests side by side.

All the above work has been focused on the engineering of desktop applications. Our approach expands this work and targets the engineering of web applications in the context of *MVWf* architectures [4, 6]. Specifically, in this paper we extend the intelligent pad architecture proposed in [7] and integrate it into a visual environment, denoted *VisualWebC*, that allows a web engineer to assemble models, views, and their relationships in a workflow process used for generating *MVWf*-based web applications.

3. Overview of the prototype

3.1 Visual constructs

The *VisualWebC* development environment we have built has several constructs that help the user efficiently develop a web application. The visual constructs depicted

in Fig. 1 are the building blocks currently available in the environment. They are divided into *local* components and *remote* components. The local components are *pages*, *textboxes*, *tables*, *labels*, and *buttons*, and they define the most used HTML elements that a web page is composed of. The *connector* object is also a local component, used to connect the other visual constructs and define their dependency relationships.

The environment also includes a *remote component* construct that is responsible for connecting the application to a running service or functionality existing elsewhere in an active web application. A remote component has one or more inputs and one or more outputs that the user must specify. The developer also needs to assign the URL address of the web page from which those services are triggered to the URL attribute of the remote component. These remote services are in essence reusable software components borrowed from existing sites and integrated in the new applications being built (note that integration is significantly different from simply linking a new web page with an existing web page, as a remote service can be tailored to the new application’s specific needs and can be graphically incorporated together with other remote services in the new application’s user interface).

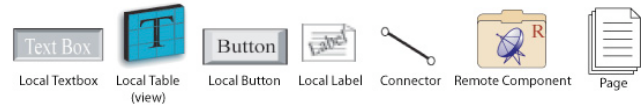


Figure 1. Visual constructs in VisualWebC

To better understand the different concepts and components present in *VisualWebC*’s framework, we will introduce the framework in the context of the following example: An investor wants to monitor stock quotes of some US companies in Japanese yen. Lycos provides a real time stock-price browsing service of US companies in US dollars. Yahoo provides a service that converts US dollars into Japanese yen based on the current exchange rate.

Fig. 2 shows the *VisualWebC*’s webpads implementation of these specifications. With these specifications in mind, we next describe more formally the common components found in *VisualWebC* and make reference to the stock monitoring example whenever necessary to illustrate the concepts.

3.2 Framework layers and components

The *MVWf*-based framework of *VisualWebC* is comprised of a four-entity hierarchy: Workflow, Model, View, and Layout (Fig. 3). For the sake of simplicity in Fig. 3 we refer to each entity in this hierarchy as *wf*, *m*, *v*, and *l*, respectively. A workflow is implemented on top of the Intelligent Pad Architecture [7], the new architecture thus created being referred to as the WebPads Architecture.

Our implementation of the *webpad* eliminates the notion of parent-child relationship that existed between

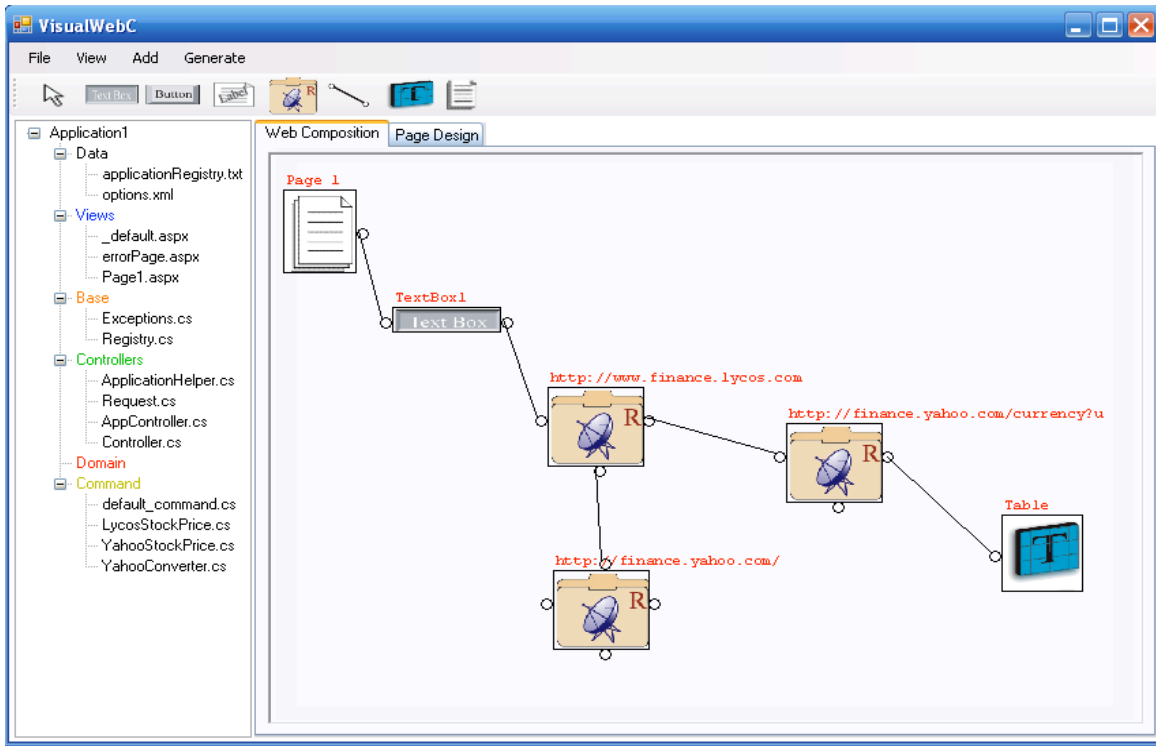


Figure 2. Sample project showing composite webpads and aliasing

different interacting *pads* in the Intelligent Pad Architecture, and replaces it with local and remote components that interact through input/output relationships.

More formally, our *MVWf*-based framework of *VisualWebC* can be described as follows: let Ψ be a remote service-oriented web application to be built with the *VisualWebC* environment. Its workflow is denoted $\Phi(\Psi) = WF = \{wf_1, wf_2, \dots, wf_n\}$, where each wf_i

$\in WF$ is a workflow process built as a *webpad* in order to develop Ψ . As depicted in Fig. 2, the application workflow is associated with a workspace called “Web Composition”. It is in this *Web Composition* workspace where developers visually construct and connect local and remote components. Each wf can be associated with two sets of entities, views $V = \{v_1, v_2, \dots, v_v\}$, and models $M = \{m_1, m_2, \dots, m_m\}$.

An example of a *view* is the *page* component “Page 1” in the *Web Composition* workspace of Fig. 2. Every $v_j \in V$ has two components, *view data* (input from the various components) and *view presentation* (local components such as a textbox or a table). The *view* entity is the main channel for user interaction with the GUI elements present within a specific view. For example, the data extracted from the remote services at Lycos (a real time stock-price in yen) and its converted US dollar value from Yahoo will be displayed as the view in “Page 1”. As depicted in Fig. 2, “Page 1” is shown in the “Views” tree node on the left hand side of *VisualWebC*’s interface as “Page1.aspx”. All $m_k \in M$ represent data sources and are drawn from the remote components in the workflow process. For example, in Fig. 2 there are three model remote components, each representing a data source (one Lycos and two Yahoo data sources). The layouts l_j of the different view representations can be managed in the “Page Design” space. A workflow process of a remote service is then

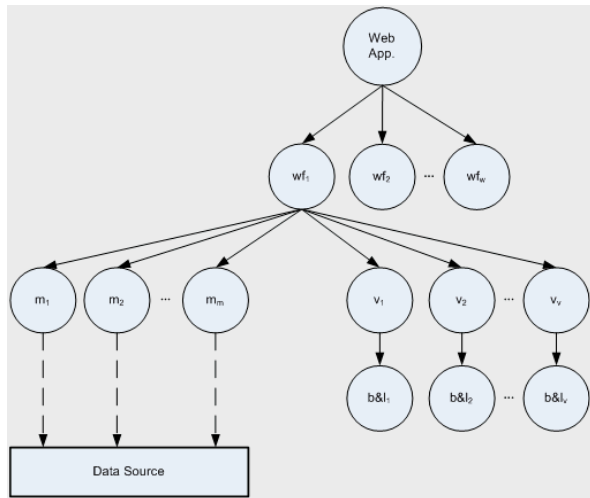


Figure 3. Abstract component hierarchy of VisualWebC

accomplished by assembling (visually constructing) a set of *model* and *view* components.

Every $v_j \in V$ and every $m_k \in M$ that are associated with a workflow $wf_i \in WF$ can have a transition rule which uses an event to determine the next view. The transition for every $v_j \in V$ and every $m_k \in M$ can be visually programmed by the user to move from one view to another (or stay in the same view). To specify possible events that might be used for transitioning between views and between models, the user is given a straightforward selection facility to choose the element that triggers the transition (e.g., “on-text-changed”, “on-failure-of-remote-data”, “on-button-click”, etc.). Cyclic data and transition dependencies are caught during the visual construction phase of the workflow.

In the current version of the *VisualWebC* prototype the state transition rules and their associated commands are captured as XML documents for each workflow. Fig. 4 shows a fragment of XML code generated to represent the relationships between the commands (transition rules) and the views for the application presented in Fig. 2. For example, in line 1 of Fig. 4, if the command “LycosStockPrice” results in a connection to the site then the status of line 2 returns “CMD_OK”, and a forward command is initiated for “YahooConverter” on line 3. The command in line 13 is then executed, and if the conversion results were retrieved properly and correctly, this command will return an “OK” status and the corresponding view in line 15 will be displayed.

```

1- <command name="LycosStockPrice">
2-   <status value="CMD_OK">
3-     <forward>YahooConverter</forward> </status>
4-     <status value="CMD_ERROR">
5-       <forward>YahooStockPrice</forward> </status>
6-   </command>
7- <command name="YahooStockPrice">
8-   <status value="CMD_OK">
9-     <forward>YahooConverter</forward> </status>
10-   <status value="CMD_ERROR">
11-     <view>errorPage</view> </status>
12- </command>
13- <command name="YahooConverter">
14-   <status value="CMD_OK">
15-     <view>Page1</view> </status>
16-   <status value="CMD_ERROR">
17-     <view>errorPage</view> </status>
18- </command>

```

Figure 4. Example of XML document generated along with the web application

A workflow of course can have more than one view. Adding different pages (views) to the workflow process can be done by attaching a new component to the workflow and specifying different inputs and outputs for it. If an input in a certain page/view leads to

an output in the same page, the user is given the choice of not linking the output to any page, meaning that the output will be automatically assigned to the same page. This functionality was added because unnecessary links may become confusing.

3.3 WebPads architecture

Knowing that we cannot plug-in and “peel” objects in a website, we extended, as previously mentioned, the Intelligent Pad Architecture [7] by eliminating the notion of parent-child relationship and replacing it with the input/output relationship, thus making *webpads* interact with each other through input/output dependencies only. Much of our WebPad Architecture in our implementation relies on the DOM-based identification of objects within HTML documents [10] and the browser’s API to reference the components, manipulate their content, and trigger their associated methods. The DOM representation structures the document as a tree and thus in order to identify a certain element in the document we need to get the path in the tree that corresponds to this element. The functionality of the remote component that allows users to wrap certain elements as input and output is based on the fact that an HTML document has an XML-like structure, consisting of several tags nested inside each other. The selection of inputs and outputs from an exiting web application is based on the DOM representation of the HTML document.

Current components in *VisualWebC*, local and remote, inherit many of their characteristics from the *Abstract component*. This design makes the process of integrating additional components (such as combo boxes, check boxes, and radio buttons) a matter of just sub-classing from the *Abstract component*.

3.4 Dependency relationships

Different types of dependencies can exist between the components that we are wrapping from different web applications. Some remote components might depend on an input from a local component while some other might depend on an output from another remote component. Let us consider in Fig. 2 Remote1, www.finance.lycos.com, and, respectively, Remote2, finance.yahoo.com/currency. Remote1 depends on an input from a local component (textbox) and Remote2 depends on the output of Remote1. Such dependencies are usually hard to track. However, our *VisualWebC*’s representation of the links between components in a workflow gives the user the ability to visualize the dependencies and ensure at run time that if an error occurred (e.g., within Remote2) its dependencies can be easily tracked and fixed. This solution also leads to

several other benefits, including: (i) visualizing the flow of information between different elements in different websites, (ii) identifying problems in case of data flow errors or dependency failure, (iii) tracking the effect of changes in the format of data, and (iv) providing the user with a way to easily add new functionalities, provided that input/output dependencies and their types are available.

4. Additional features

To make *VisualWebC* more useful we added several features aimed at optimizing the generation of the web application and extending its lifetime. The Dynamic Link Library (DLL) we created minimizes the code that needs to be generated by *VisualWebC*. In addition, the mechanism of creating a website is implemented through a series of steps whose purpose is to define the dependencies between different remote components and the local components of the website under construction. By using the DLL, the time needed to generate the web application is reduced. Furthermore, our environment supports multiple types of inputs and outputs that cover a large portion of the possible components that need to be taken from remote applications. Finally, we have appended the aliasing mechanism in order to make such websites more resilient to future changes in addressed remote objects.

4.1 Triggering remote web services and generating results

To handle the issues of triggering remote web services and generating results, we designed a component implemented as a DLL that takes the input and the URL address of a certain website and returns the output as specified by the user. This design solution ensures that the code responsible for connecting to remote web applications will be reused for each application, which in turn reduces the overhead of regenerating this code for every remote component. By using this DLL component, our framework is only responsible for generating the code that will use this library and perform the functionalities required. This library encapsulates the concepts of the *webpad* and is responsible of simulating its behavior.

4.2 Different types of inputs and outputs

We have devised a method for extracting the output of a remote functionality (service) as a list of results. A good example to illustrate this feature is a search engine where the user needs to specify all the results as output in order to display all of them on his or her web page. Thus, in order to reduce the overhead, we provided the user with the facility to retrieve the list of

results by specifying the output of the component as of type list, ask him or her to wrap only the first two results, and to specify the number of results that need to be wrapped for output. Our implementation internally applies a string-matching algorithm in order to retrieve the difference in the HTML paths of the results, use this difference to compute the HTML path of the remaining results in the list, and address the remote elements with computed HTML paths.

Currently, our prototype handles several types of inputs and outputs. An input/output can be almost any element of a website, including a text field, button, text area, text, and so forth. However, unsupported elements can be very easily integrated.

4.3 Aliasing of remote components

To deal with the possibility of having failed remote services, the web developer is given the option to specify one or more aliases for a remote service in a webpad. The alias is a website that provides a similar functionality as the original remote component. This aliasing mechanism is left optional for the developer. In addition, the developer can specify as many aliases as needed in an effort to reduce the number of possible “out of service” or “error” occurrences. The generation of an alias is partially automatic to assist the developer and eliminate the overhead of linking, creating, and re-initializing the alias as the original. As we have noticed in our experimentation with the framework, this approach minimizes the errors that might occur during the re-initialization process. An example of such an error is specifying different numbers of inputs or outputs. For example, since Yahoo Finance also provides a real time stock-price browsing, we added it as an alias to the Lycos Finance. Thus, in case Lycos is down then the stock price will be taken from Yahoo Finance instead and will be passed to the currency converter. As depicted in Fig. 2, the output of Lycos Finance (or its alias) will be the input for Yahoo Currency Converter. The output that will finally be displayed in “Textbox 1” of the current view (Page 1) will be that of the currency converter which will be the price of the stock in yen.

4.4 Importing page design

The generated page/view has a very simple design – the elements are just displayed on the web application in an ordered manner. However, we wanted to provide a richer display, especially for our experimental studies, so we provided the users with a way of importing an existing page (HTML or ASP), loading it into our system, and use it when generating the web

application. Once the user imports a page (with some design), our prototype then uses the DOM representation of the page to identify the different components of the page (textboxes, button, etc.) and a new page component will appear in the *Composition Window*. The user then is required (fact indicated by a blinking page icon) to link this page to existing local components in order to identify their relationships.

4.5 Generated website model

When the user imports a certain page, this will not affect the model of the website that is generated because an imported page only affects the display of the elements. Also, if the page had already different functionalities they will not affect the generated model because they will remain untouched.

Along these advantages our approach still has its breaches and disadvantages, as follows. First, if a remote website made a change in its design then all the paths of the inputs and outputs will change consequently and our solution will malfunction. Also, if changes were done to the format of a wrapped component's content, this might lead to erroneous behavior if this wrapped component was defined as an input to another remote component. Such a change will cause a type mismatch and the system will not function properly. Finally, in order to function correctly, the website that relies on services from other websites will need to open these websites before being fully loaded, which may require more time than usual and thus make the new website operate somehow slower. Yet, in our approach we relied on the internet continuously improving and its bandwidth steadily increasing.

5. Experimental studies

To investigate the effectiveness of our tool and guide our future work, we conducted two short studies (early indicators of our approach's performance) to answer the following questions:

1. How effective is our tool in the overall production of remote service-oriented web applications in development teams that were educated on the concepts of programming in *MVWf*-based frameworks but not in those of *VisualWebC*?
2. How effective is our tool in the overall production of remote service-oriented web applications in development teams that were not educated on the concepts of programming in *MVWf*-based frameworks but were given a full tutorial on the use of *VisualWebC*?

The first question was intended to help us investigate if our engineering technique and tool are in line with the *MVWf*-based web engineering paradigm.

The second question was intended to help us identify if a working knowledge of the *MVWf*-based programming paradigm is a prerequisite for using the *VisualWebC* tool.

To answer the first question, we gathered a group of students ("Group A") that consisted of 3 teams, each team being composed of 3 students of different programming skills. More precisely, each team included an advanced programmer (graduate student), an intermediate programmer (senior undergrad), and a beginner programmer (sophomore undergrad), none of them with industrial experience in web engineering or the *MVWf*-based web development paradigm. We then conducted a training session to introduce Group A (teams A1, A2, and A3, with a total of 9 students in the group) to the essential concepts of programming in *MVWf*-based frameworks. This session proved to be the longest and most difficult, as students in all teams had various difficulties in creating a mental image of the workflow process. To assist in this step, we used the Ruby-on-Rails framework [6] as means to deliver hands-on tutorials on the *MVWf* paradigm.

The teams were then asked to implement, using *VisualWebC*, a web application that, once given the capital of a certain country, retrieves the following information:

- The 6-day weather forecast in that capital;
- Headlines of well-known international newspapers containing the capital's name;
- A list of hotels in the capital.

For each team, we recorded the relative correctness (checked against program requirements) of their results and the time it took them to announce that they have fulfilled the specifications. The data for correctness, evaluated by the instructor and two graders, and time needed in Group A teams are shown in Table 1.

To answer the second question, we assembled a group of students ("Group B") with three teams and three students in each team with the same skill distribution as in Group A. We next conducted a learning session to introduce Group B to the concept of building remote service-oriented web applications using the *VisualWebC* environment. This session was by far shorter than the first. We then asked the second team to implement using *VisualWebC* the same web application specifications described above. The data for correctness and time needed in Group B teams are shown in Table 2.

The results of the first study confirmed our expectation. The teams in Group A managed to get a very good percentage in the correctness of the overall solution of the problem, but it took them an average of 33.7 minutes to complete the task. This indicates that our programming methodology follows the *MVWf*-based programming methodology.

The results of the second study were somewhat surprising since they showed a lower average of correctness as compared with Group A. However, notably, the teams in Group B managed to get their task done in less than half the time needed by Group A to finish the same task. The lower percentage in correctness suggests that a good knowledge on the *MVWf*-based programming methodology is important. Having said that, however, we should note that the information collected from Group B on *MVWf*-based programming after the experiment showed that through the use of our visual tool the group managed to get a good understanding of the paradigm.

Table 1. Results of the first study (group with working knowledge of *MVWf* paradigm)

	<i>VisualWebC</i> Tutorial	<i>MVWf</i> Tutorial	Average correctness	Average time
Group A Average	No	Yes	97.7%	33.7 min
Team A1	No	Yes	97.5%	39.0 min
Team A2	No	Yes	100.0%	37.0 min
Team A3	No	Yes	95.5%	25.0 min

Table 2. Results of the second study (group with working knowledge of *VisualWebC*)

	<i>VisualWebC</i> Tutorial	<i>MVWf</i> Tutorial	Average correctness	Average time
Group B Average	Yes	No	94.0%	14.7 min
Team B1	Yes	No	97.5%	15.0 min
Team B2	Yes	No	89.0%	17.0 min
Team B3	Yes	No	95.5%	12.0 min

Although the tests were intended only to give an indication of the proposed approach’s performance and evolved empirical studies are still needed, the data collected from both experiments suggested that we should continue to pursue the idea of creating a visual environment that relies on the functionalities of a *MVWf*-based programming system such as Ruby-on-Rails [6] or Bricks [13] and augment them with our WebPad-based approach.

6. On the WebPads-based approach, agility, and the SPLE context

As Pohl, Böckle and van der Linden indicate, “software product line engineering has proven to be the methodology for developing a diversity of software products and software-intensive systems at lower costs,

in shorter time, and with higher quality” [14]. As pointed by the authors, one of the key aspects of this modern methodology is its focus on *reuse* and on creating *software platforms* for application development that emphasize well-managed and flexible reuse of *software components*, supportive to adequately dealing with inherent product variability.

Another defining characteristic of Software Product Line Engineering (SPLE) is represented by a key dual software process, consisting of *domain engineering*, which establishes and realizes the commonality and the variability of the product line, and *application engineering*, which draws on the strengths of the product line and builds software applications that reuse common domain artifacts while also ensuring that required product variability is satisfied [14].

Furthermore, due to the specifics of following the above two complementary, powerful software processes and the goal of properly managing software product variability SPLE is concerned with ensuring proper *documentation* of product variability and created artifacts, including requirements, design, realization, and test artifacts [14].

While operating essentially as agile (quick, flexible, adaptable, user-oriented) instruments in the realm of engineering web applications, which in itself represents an interesting area of exploration for SPLE, the *WebPads*-based approach introduced in this paper and its supporting tool, *VisualWebC*, offers excellent examples of support for *software reusability*. After all, the remote services incorporated in new applications are remarkable illustrations of reusing software components, be they weather forecast modules, stock evolution monitors, shopping carts, online unit converters, or keyword-based searching mechanisms.

In terms of *domain engineering* and *application engineering*, one can envision applying *WebPads* and tools similar to *VisualWebC* to building web applications using the systematic process directions of the SPLE methodology, combined with (intrinsic to webpads) elements of development agility. Our approach being organized around easily modifiable, reconfigurable and reusable views, models, workflows, layers, and webpads (web services), it is very likely that it can effectively support both the common artifacts of a web development “domain” (e.g., the on-line shopping “domain”) and the particular aspects of a specific application in that “domain” (e.g., on-line shopping of automotive parts, such as tires or wheels).

Finally, *documentation* and, in general, *process* and *product organization*, all important to SPLE, could significantly benefit from using easy-to-built and efficient web-based solutions for collaborative software product development made possible by the proposed *WebPads*-based approach (e.g., web-based

error-tracking tools, jointly updated product data dictionary, and online tools for project progress reporting, assessment, and prediction).

In summary, numerous interesting directions of research and development are offered by the prospect of placing *WebPads*-based applications and tools (the latter, very agile in nature) under the SPLE umbrella.

7. Future work and conclusions

Our approach presents several significant advantages. First, it allows the creation of web applications with complex functionality with relatively little effort and time. Importantly, it allows novice programmers to build websites easily and reduces the cost and time of creating such websites.

Second, the use of trusted, well tested, permanently updated, and secure services allows the newly generated web applications to inherit these features and gain the trust of users, who do not have to worry about the security of the remote components they rely on.

Third, testing a new web application is reduced to testing the inter-services relationships that exist between different wrapped web applications rather than testing the intra-services relationships that characterizes each contributing application (because the latter are already tested by their providers).

Fourth, if one of the remote web applications updates its database or enhances its mechanisms of providing the outputs, these changes will be directly reflected in the new web application, since it only depends on the paths of the inputs and outputs. Consequently, there is no need to worry about updating the web application, especially because the connection to the remote services is established dynamically.

Regarding future work, a problem that we plan to tackle in the near future is to implement a proxy class between the website created and the websites referenced in order to minimize the effects of changes in their data formats. Specifically, when a referenced website changes the format of its inputs and/or outputs an interface is needed between the two applications to accommodate the changes.

Also part of future work, we are interested in developing a more rigorous model using client/server agents that will track changes made on the remote components and refresh the local components, thus keeping the information always up-to-date.

Finally, as discussed in the previous section, placing the naturally agile (fast, flexible, service-oriented, reconfigurable, easily manageable, user-friendly) *WebPads*-based web development approaches and tools in the scope of the SPLE methodology offers many promising directions of future exploration and expansion. A larger example of web application

development using the *WebPads* concept, the *VisualWebC* tool, and the methodological directions provided by SPLE is also part of our future work.

Acknowledgments

We would like to thank the students at AUB for their collaboration and patience in the experimentation sessions. Our thanks go also to ACM SIGCHI for allowing us to modify templates they developed.

References

- [1] Handschuh, S., Staab, S. and Volz, R. "On Deep Annotation," *Procs. of WWW-2003*, Budapest, 2003.
- [2] Handschuh, S. and Staab, S. "Annotation of the Shallow and the Deep Web," In Handschuh, S. and Staab S. (eds.), *Frontiers in Artificial Intelligence and Applications*, vol. 96, IOS Press, Amsterdam, 2003, pp. 25-45.
- [3] Hohpe, G. "Developing Software in a Service-Oriented World," white paper, 2005. Accessed April 15, 2006 at www.enterpriseintegrationpatterns.com/docs/
- [4] Karam, M., Keirouz, W. and Hage, R. "An Abstract Model for Testing MVC and Workflow Based Applications," *Procs. of the 2006 IEEE Intl. Conference on Internet and Web Applications and Services (ICIW'06)*, 2006, pp. 206-211.
- [5] Leff, A. and Rayfield, J.T. "Web-Application Development Using the Model-View-Controller Design Pattern," *Procs. of the 5th IEEE Enterprise Distributed Object Computing Conference*, 2001, pp. 118-124.
- [6] Ruby on Rails, "Web Development that Doesn't Hurt," accessed April 20, 2006 at <http://www.rubyonrails.org>
- [7] Ito, K. and Tanaka, Y. "A Visual Environment for Dynamic Web Application Composition," *Proceedings of the 14th ACM Conference on Hypertext and Hypermedia*, 2003, pp. 184-193.
- [8] Ito, K. and Tanaka, Y. "Visual Wrapping and Functional Linkage of Web Applications," *Proceedings of the Workshop on Emerging Applications for Wireless and Mobile Access*, Budapest, Hungary, 2003.
- [9] Tanaka, Y., Fujima, J. and Sugibuchi, T. "Meme Media and Meme Pools for Re-editing and Redistributing Intellectual Assets," *LNCS-2266*, 2003, pp. 28-46.
- [10] W3C Document Object Model (DOM) accessed May 23, 2006 at www.w3.org/DOM
- [11] Tanaka, Y. *Meme Media and Meme Market Architectures: Knowledge Media for Editing, Distributing, and Managing Intellectual Resources*, John Wiley & Sons, 2003.
- [12] Fujima, J., Lunzer, A., Hornboek, K. and Tanaka, Y. "Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access," *Procs. of the 17th ACM Symposium on User Interface Software and Technology*, 2004, pp. 175-184.
- [13] Bricks, "Lightweight Web Application Development Framework," accessed June 9, 2006 at <http://www.pythonweb.org/doc/bricks.doc.html>
- [14] Pohl, K., Böckle, G., and van der Linden, F., *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer, 2005.