ENABLING AN END-USER DRIVEN APPROACH FOR MANAGING EVOLVING USER INTERFACES IN BUSINESS WEB APPLICATIONS

A Web Application Architecture using Smart Business Object

Xufeng (Danny) Liang and Athula Ginige
School of Computing and Mathematics, University of Western Sydney, Sydney, Australia

Abstract: As web applications become the centre-stage of today’s businesses, they require a more manageable and holistic approach to handle the rapidity and diversity with which changes occur in their web user interfaces. Adopting the End User Development (EUD) paradigm, we advocate an end-user driven approach to maintain evolving user interfaces of business web applications. Such an approach, demands a complementary web application architecture to enable a flexible, managed, and fine-grained control over the web user interfaces. In this paper, we proposed a web application architecture that embraces a dedicated UI Definition Layer, enforced by a web UI Model, for describing changes in the user interfaces of business web applications. This empowers business users to use intuitive web-based tools to effortlessly manage and create web user interfaces. The proposed architecture is realised through the Smart Business Object (SBO) technology we previously developed. We have also created a toolkit based on our proposed architecture. A tailored version of the toolkit has been utilised in an enterprise level web-based workflow application.

1 INTRODUCTION

The ancient Greek philosopher Heraclitus wrote that “the only constant is change”. Today, the challenge in developing business web applications is the shift from implementing functionality to managing complex changes. One of the elements that often get changed is the web user interfaces (UIs). Business web applications may consist of hundreds of screens. The intensiveness and degree with which change occurs in the user interfaces of business web applications demands a more flexible approach to handle its constant evolution.

In practise, it is difficult to precisely capture all the user interface requirements during the requirement analysis phase of a project. One factor causing this is the uncertainty and lack of understanding business users have about what they need without being able to see or use the final software artefact. Another factor is the different ways in which the developed web application is used across different departments of an organisation. For instance, the management departments in universities often use the term AOU (Academic Organisation Units) to refer to what most academic staffs refer to as schools and colleges. Consequently, we often overlooked the perspectives from potential user groups of a web application, and end up with a user-unfriendly application with low usability. Once the web application has been deployed, business users may see new opportunities or simply find certain obsolete information in the web user interfaces that, for example, does not comply with the latest government regulations. Thus, amendments to those web user interfaces become necessary again.

The conventional approach is to call the project development team to implement the changes. However, it is common that before the changes are implemented or re-implemented, requirements have changed again. To put an end to this never-ending cycle, requires a radical change to the conventions in handling the constant evolution of the user interfaces in business web applications. The End-User Development (EUD) paradigm advocates that software should not only be easy to use, but also easy to develop (Lieberman et al., 2006). Empowering end-users and allowing trained end-users to maintain or even enhance existing appli-
ENABLING AN END-USER DRIVEN APPROACH FOR MANAGING EVOLVING USER INTERFACES IN BUSINESS WEB APPLICATIONS - A Web Application Architecture using Smart Business Object

cations is a cost-effective way to support web application evolution (Wulf and Jarke, 2004). From our experience in developing web applications for enterprises, there is a growing demand from business users to maintain the user interfaces of their web applications. Thus, we believe that embracing the End-User Development paradigm and empowering business end-users to manage web user interfaces is a practical and effective approach. From the organisation’s perspective, this approach translates directly into cost reduction in IT maintenance while encouraging user retention and ownership of the web application.

The distinctive advantage of such an end-user driven approach is that it enables a hybrid, parallel mode of developing web applications. With the help of contemporary model-driven development tools, professional web developers can quickly generate (and release) the intended web application with basic functionality and reasonable user interfaces. With the appropriate tools, business end-users can then constantly customise the web user interfaces until they are satisfied, while professional developers can continue to develop and finalise the remaining components of the web application. This means that both professional developers and business end-users can work on the intended web application at the same time. Therefore, an end-user driven approach to managing evolving web user interfaces can also compress the timeline required for developing business web applications.

An end-user driven approach, however, requires intuitive tools for managing web user interfaces. Accordingly, a web application architecture that supports the realisation of such tools is desirable. Such tools should empower business end-users to apply changes to the web user interfaces dynamically without affecting the web application running, in order to eliminate the need for interventions from professional developers.

In end-user driven, WYSIWYG-type web tools, such as Click (Rode et al., 2005), web UI changes are made on the basis of a screen (page). However, taking each screen as a unit for managing change has its shortcomings. In business web applications, it is common to have web user interfaces, such as web forms, that span across multiple screens for presenting complex business objects with a large number of attributes. Thus, in order to empower business end-users to manage complex web user interfaces in business web applications, a more manageable and sensible way of partitioning web user interfaces is required.

It is also vital for business end-users to be able to customise every UI element (controls, widgets) of their user interfaces while preserving a consistent “look and feel” across the web application. This implies that the underlying web application architecture should support a fine-grained, yet controlled approach for managing web user interfaces.

Existing web modelling approaches, such as UWE (Koch and Kraus, 2002), WebML (Ceri et al., 2000) and OO-H (Gomez et al., 2001), focus mainly on the creation aspect of web user interfaces with via detailed navigation modelling and presentation modelling. However, the evolution aspect of web user interfaces is often neglected. These techniques require extensive navigation and presentation models to be defined upfront before web user interfaces can be generated. While using conceptual models to maintain and recreate (regenerate) web user interfaces is a plausible approach for professional developers to maintain web user interfaces, it is impractical for end-users. Business end-users do not have the skill-set to understand and directly use complex conceptual modelling techniques, such as UML, to manage web user interfaces. (Atterer, 2005) found that concrete navigation modelling and presentation modelling can make the development process “work-intensive” and increases complexity even for professional developers. On the contrary, our proposal is end-user driven with a strong focus on the evolution aspect of web user interfaces. Our approach eliminates the need for extensive navigation and presentation modelling prior to the generation of web user interfaces. The underlying UI Model supporting our proposed architecture is concealed by intuitive form-based web tools, thus, making it completely transparent to business end-users.

Our proposed architecture is built on our previous work of the Smart Business Object (SBO) (Liang and Ginige, 2006). SBO is a framework for lightweight modelling and rich web presentation of business objects. It enables Agile Model Driven Development (AMDD) (Ambler, 2003) of business web applications. It allows business object to be modelled and generated using a lightweight, near-English syntax modelling language call SBOML (Smart Business Object Modelling Language) (Liang and Ginige, 2006). Most importantly, SBO is capable of auto-generating a rich collection of highly customisable web UI Components, such as web forms, tables, menus, and charts, for the modelled business objects. On the one hand, SBO is smart enough to automatically deduce and generate the best suitable web user interfaces for the modelling business objects with zero configurations, eliminating the need for extendible user interface modelling. On the other hand, SBO makes provision for fine-tuning the generated...
web user interfaces when necessary using high-level, compact data structures. Moreover, the web user interfaces generated by SBO are “self-contained”, allowing a complete web user interface to be partitioned in a sensible way for business end-users to manage. These characteristics make SBO a suitable candidate for the implementation of our proposed architecture and the realisation of an end-user driven approach to manage web user interfaces.

We will first introduce the overall architecture, the UI Definition Layer, as well as the required UI Model. We will then explain how the SBO facilitated our proposed architecture. Lastly, we will demonstrate the toolkit we have developed based on our proposed architecture, and its utilisation in an enterprise level web-based workflow application.

2 THE PROPOSED WEB APPLICATION ARCHITECTURE

In most existing web applications, data are directly bound to disparate UI elements (controls, widgets), such as selection lists, buttons, and hypertext links, and rendered as web user interfaces using web templates or server-side/client-side scripts. While this is an efficient approach, it is also inflexible for changes, since every single change made to the web user interface requires updating the corresponding web templates or scripts directly. In order to achieve greater flexibility, we propose a web application architecture that utilises a dedicated UI Definition Layer for describing the changes in web user interfaces. Figure 1 compares existing web applications on the left with web applications using the desired UI Definition Layer on the right.

It is difficult for business end-user to manage disparate UI elements (controls, widgets) scattered across different screens (pages) of the web application. Therefore, we need to partition complex web user interfaces into discrete UI Components, such as web forms, tables, and charts, such that business end-user can easily relate them to their representations in the real world. Each UI Component is a logical composition of many UI elements. In this ways, web UI changes are made as per UI Component, as oppose to per screen.

UI Components must be “self-contained” and standardised. They have both rendering and event handling capabilities (to handle various user actions). The relationship between each UI component type and various UI elements is defined by a UI Model.

The UI Model serves as the “gatekeeper” standardising the overall structure of each UI Component. Accordingly, the UI Model should be maintained by professional developers to make provision for the defining new UI Components or upgrading existing ones.

In order to achieve a fine-grained control over the generated web user interface, every UI elements of a UI Component must be customisable by end-users. Notably, UI Components may contain other UI Components. For example, a web table can delegate the action for updating individual records to a web form component. Figure 2 is an abstract view showing the structure of the UI Model through several commonly
used UI Components.

In essence, the proposed UI Definition Layer is a set of UI Definitions. Each UI Definition is an instance of the UI Model. An example of a simple UI Definition implemented using XML would be:

```xml
<ui_def version="1.0" encoding="UTF-8" ?>
  <form_title>Add New Employee</form_title>
  <form_description>Please fill in the necessary information about the employee</form_description>
  <fields>
    <field name="address">
      <label>'Home Address'</label>
      <required>1</required>
      <comment>Please enter your home address here with Post code</comment>
    </field>
  </fields>
</ui_def>
```

UI Definitions are compact. They are designed for customising necessary UI elements rather than defining each UI element that should exists in a web user interface. In other words, UI Definitions does not define a web user interface but only the changes required for that web user interface. This is a unique characteristic that differentiates our UI Definitions, as well as our overall approach in generating web user interfaces from existing declarative user interface definition languages such as UsiXML (Vanderdonckt, 2005) and UIML (UIML, 2007). In the previous UI Definition example, it does not mean that the generated web user interface (the actual web form) will consist only one field called address. What it does means is that while enforcing special requirements for the form’s title, description, and the address field, display other UI elements (such as form fields if exists) using the default settings automatically. In reality, the definition of the modelled business object (e.g. an employee has first name, last name, address, phone number) plus the intrinsic characteristics of the intended UI component (e.g. forms have fields, pie charts are round) is sufficient to derive and render the actual web user interface with default behaviours. In this case, a UI Definition can be an empty placeholder reserved for future changes. We will explain this more with the introduction of the SBO’s rendering APIs in Section 3.

It is also imperative for UI Definitions to be implemented using high-level, lightweight data structures, such that tools can be easily built on top in order to support business end-users in managing their web user interfaces.

Rendering a UI Component, i.e. generating the actual web user interface, is the binding of the UI Definition and the referenced business object (data object). Thus, the complete web user interface presented to the users is a series of rendered UI Components controlled by a master web template for layout. The overall process is illustrated in Figure 3.

![UI Composition Process](image)

**Figure 3: UI Composition Process.**

3 THE ENABLING TECHNOLOGY

SBO is built on top of existing ORM (Object Relational Mapping) technology. SBO allows web-
enabled business objects to be modelled using a lightweight, near-English syntax modelling language called SBOML (Liang and Ginige, 2006). Modelling using SBOML is streamlined via the SBO Builder tool we have previously developed (Figure 4). Modelling business objects using SBOML implies the auto-generation of:

- The underlying databases (and table structures) for object persistence,
- The necessary ORM classes for manipulating the modelled business objects using Object Oriented Programming techniques, and
- A rich set of web user interfaces for presenting the modelled business objects to the web.

Writing code to render web user interfaces is a tedious and laborious task. Database schema does not provide the sufficient semantics required for presenting business objects onto the web. Thus, in order to derive the most suitable web user interfaces for the modelled business objects, SBO maintains a global schema that represent the additional aspects, such as presentation logic, validation logic, and content handling mechanism of the business objects. Such schema makes provision for, e.g.:

- Auto-generating the validation logic required for the email attribute during editing (often via web forms) and showing it as a mailto hypertext link by default in view operations
- Rendering the photo attribute as images on the web browser for view operations while providing the necessary uploading facility for editing by default
- Displaying the address attribute as a map automatically on view operations (as shown in Figure 6).

Apart from the global schema, SBO has three main components. The Builder component is a SBOML parser and a business object run-time loader. The Metaobject component maintains the customisable properties of the modelled business objects as well as the relationship among them. The main interest of this paper is its third component, called Renderer, which is responsible for auto-generating web user interfaces. Essentially, Renderer is a set of rendering APIs. As previously mentioned, UI Definitions are aimed for describing the necessary changes for the web user interface. Thus, a mechanism is indispensable for generating web user interfaces without the need for defining the complete web user interfaces and, at the same time, being able to fine-tune the generated web user interfaces if necessary. The SBO’s rendering APIs are capable of fulfilling this requirement. For example, to render the web form in Figure 5 for an “employee” business object requires a single line of code by executing the render_as_form API:

```
organisation::employee->render_as_form();
```

SBO’s rendering APIs require no declaration or configuration prior to the generation of a web user interface. As illustrated in Figure 5, we do not need to specify the existence of the required fields, their order, validation, as well as localised presentation, i.e., we also do not need to declare that:

- `first name`, `last name`, etc. are mandatory fields,
- all fields should be validated accordingly,
- `photo` is an upload field,
- a calendar (date picker) is required to assist data entry for the `date of birth` field and formatted according to the users’ locale

since SBO is able to automatically derive all these information from the underlying data model (the database schema), the Metaobject component, as well as the definitions stored in the SBO’s global schema.

We can use the following line of code:

```
organisation::employee->render_as_table(
ajax => 1, create => 1, edit => 1, delete => 1);
```

to generate an AJAX-enabled web table with create, edit, and delete access to the “employee” business object shown in Figure 6.

Similarly, SBO’s rendering APIs supports a fine-grained control over the generated user interfaces through various parameters. An example would be customising the previous “employee” web form in Figure 5 with the following requirements:

- Set the title of the form to “Add New Employee”,
Add a description to the form as follows: “Please fill in the necessary information about the employee”.

For the address attribute, change the form field’s label to “Home Address”, make it a mandatory field, and add a tool-tip using the following information “Please enter your home address here”.

Display only the first name, last name, email, address, and position fields in that order, and

Hide the cancel button

To achieve this, all we need is to pass the following parameters (in the form of associated array) directly to the render_as_form API:

```php
organisation::employee->render_as_form(
    form_title => 'Add New Employee',
    form_description => 'Please fill in the necessary information about the employee',
    fields => [
        name => 'address',
        label => 'Home Address',
        required => 1,
        comment => 'Please enter your home address here with Post code',
    ],
    field_order => ['first_name', 'last_name', 'email', 'address', 'position'],
    hide_cancel => 1);
```

Once the above code segment is executed, the web form shown in Figure 7 is generated. In this way, UI Definitions can be easily implemented by serialising those parameters. In run-time, UI Definitions are retrieved, inflated (deserialised), and passed back to the corresponding SBO APIs to generate the required web user interfaces.

Notably, the rendering APIs are architected towards the MVC (Model View Controller) design pattern. The model is determined by the structure of the calling business object. The generated view of each rendering API is template-driven. Custom controllers can be also specified to the rendering API to handle special actions. Consequently, each rendering API is “self-contained” due to its rendering capability and event handling capability. This “self-contained” nature of the rendering APIs makes provision for partitioning of web user interfaces as the required UI Component concept in our proposed architecture. Therefore, using SBO, a complete web user interface presented to the users can be considered as a composition of one or many UI Components, generated by rendering APIs, governed by a master web template(s) for layout. The master web template(s) also serves the purpose for retaining a global “look and feel” of the web application, as well as incorporating other static UI elements, such as simple text descriptions, outside the UI Components.

Figure 8 is an example showing the rendering of two UI components: a bar chart and a web table, along with a simple static heading, in a master web template.

The “self-contained” nature of the rendering APIs also implies an implicit navigation model within the generated web user interfaces. For instance, in Figure 6, once an user clicks on the “Create Employee” link, the web table will hide and a form is then shown.

“Look and feels” can be achieved by specifying customised web template(s) to the corresponding rendering API. Therefore, professional developers can ensure the overall “look and feel” of the web application by maintaining the web templates used by the rendering APIs.

Rendering APIs usually have one or more built-in controllers for handling user actions. In the previous “employee” web form, when the “create” button is clicked, the same rendering API can handle the submission (post) of the form, including uploading the binary image file. Additional coding is not required since there is a built-in controller for handling such “create” action. Custom controllers can be also specified to the rendering API to handle special actions.

Consequently, each rendering API is “self-contained” due to its rendering capability and event handling capability. This “self-contained” nature of the rendering APIs makes provision for partitioning of web user interfaces as the required UI Component concept in our proposed architecture. Therefore, using SBO, a complete web user interface presented to the users can be considered as a composition of one or many UI Components, generated by rendering APIs, governed by a master web template(s) for layout. The master web template(s) also serves the purpose for retaining a global “look and feel” of the web application, as well as incorporating other static UI elements, such as simple text descriptions, outside the UI Components. Figure 8 is an example showing the rendering of two UI components: a bar chart and a web table, along with a simple static heading, in a master web template.

The “self-contained” nature of the rendering APIs also implies an implicit navigation model within the generated web user interfaces. For instance, in Figure 6, once an user clicks on the “Create Employee” link, the web table will hide and a form is then shown.
There are several ways to customise or enhance the default navigation behaviours:

- Customise the parameters passed to the corresponding rendering API, such as whether to hide or show a web form on certain events
- Specify custom web templates to the corresponding rendering API to incorporate additional navigation paths or structures
- Define custom controllers to trigger the rendering of another web user interface, in order to sequenced user interfaces.

Due to the scope of this paper, we will not focus on the details of customising the implicit navigation model of the rendering APIs.

By harnessing the SBO framework, our proposed web application architecture is shown as Figure 9. As previously mentioned, SBO is built on top of ORM technology to achieve object persistence. The SBO layer facilitates a Renderer component for generating a rich portfolio of web UI Components for the modelled business objects. The proposed UI Definition Layer in this paper, is built on top of the SBO layer for flexible and fine-grained control over the web UI Components. A UI Generation Tool is utilised for business end-users to manage User Interface Definitions in the UI Definition Layer. The final user interfaces of the web applications are generated through the UI Definition Layer using SBO’s rendering APIs. We will demonstrate the UI Generation Tool in the next section.
CONCLUSION AND FUTURE WORK

Web user interfaces are always evolving. To achieve an end-user driven approach to manage user interfaces in business web application, we introduced a web application architecture with a dedicated UI Definition Layer for describing changes in web user interfaces. The UI Definition Layer takes advantage of the SBO framework for generating highly customisable, “self-contained” web UI Components. We have created a toolkit to demonstrate the utilisation of our proposed architecture.

A customised version of the toolkit is employed by OCAS (Online Course Approval System) (University of Western Sydney, 2006), an enterprise level web-based workflow system for automating the courses and units (subjects) approval processes in University of Western Sydney. Figure 13 is only a snapshot of the OCAS application.

Courses and units are the core business functions of a university. The sophisticated approval processes for course or unit involve almost every departments of the university. One of the main challenges in developing the OCAS application was that the web user interfaces keep evolving. This was due to many reasons, for example:

- Despite the fact that we have interviewed nearly every departments in the university, it was still impossible for us to capture all the UI requirements to satisfy all the user groups of the application.
- Some required UI specifications, such as help information required by the UIs for explaining specific jargon, does not even exist in any current form of documentation.
- The specification of the underlying business objects (some with more than 150 attributes) were also evolving. As a result, the corresponding web user interfaces need to be frequently changed accordingly.
- Some information place on the developed web user interfaces became obsolete even before the
We cannot change the fact that the web user interfaces are constantly changing. Traditional approaches were inappropriate for developing the OCAS application. Consequently, we have embraced EUD and utilised our proposed architecture to develop OCAS. We have tailored our toolkit to specialise in managing all the descriptive text, such as form descriptions, form field labels and tool-tip information, as well as table column headings, for the web user interfaces of the application (Figure 14). SBO is used to generate all the web user interfaces and entire data model for the application is modelled using SBOML. Once the basic web application is released, trained business end-users could then use the tailored toolkit to customise and finalise the web user interfaces, while the development team was implementing other core components of the application. The OCAS application was a parallel development between professional developers and business end-users. In such way, the project’s development timeframe was greatly reduced.

Further investigation on the usage pattern of the toolkit is desirable to analyse its effectiveness from the perspective of business end-users.

REFERENCES


University of Western Sydney (2006). Online course approval system (ocas).
